UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze
Matematiche Fisiche e Naturali

Corso di Laurea in
Scienze dell'Informazione

Tesi di laurea

Un linguaggio di interrogazione e un ottimizzatore per interagire con una rete di sensori

Relatori:

Dott. G. Amato

Prof. S. Chessa

Controrelatore:

Prof. G. Ghelli

Candidato:

Letizia Alfonsa Ciaccio

Anno Accademico 2005-06

Un linguaggio di interrogazione e un ottimizzatore per interagire con una rete di sensori

Letizia Alfonsa Ciaccio

Dipartimento di Informatica
Università di Pisa

Sommario

Le reti di sensori senza fili, composte da nodi autonomi ciascuno provvisto di batteria, sensori, antenna radio, processore, memoria, timer, stanno trovando innumerevoli applicazioni in molti campi.

In questa tesi è stato definito un linguaggio per interrogare una rete di sensori senza fili similmente a come si farebbe per un database tradizionale. Nella tesi è stato realizzato il parser delle query e la generazione del piano di esecuzione distribuito fra vari nodi, secondo un'algebra opportunatamente definita. E' stato progettato e realizzato l'ottimizzatore delle query. La particolarità della strategia di ottimizzazione è dovuta al fatto che si cerca di minimizzare il consumo di energia nella rete di sensori.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Negli ultimi anni lo sviluppo delle tecnologie hardware ha reso possibile la realizzazione di dispositivi sempre più potenti e miniaturizzati.

Già oggi sono disponibili sul mercato piccoli dispositivi autonomi che occupano poche decine di centimetri cubi (vedi par. 2.1) e, secondo la legge di Moore, in un futuro prossimo avremo dispositivi che misureranno meno di un centimetro cubo e saranno abbastanza potenti da eseguire sistemi operativi standard (ad es. versioni adattate di Linux), mentre il loro costo sarà molto contenuto.

Questo progresso tecnologico, insieme a quello riguardante le comunicazioni senza fili, ha costituito la base per nuovi tipi di applicazioni, tra cui le reti di sensori. Una reti di sensori, detta anche *Wireless Sensor Network* (*WSN*), è una rete distribuita che consiste di un grande numero di nodi ciascuno provvisto di strumenti di misurazione di parametri ambientali, capacità di computazione e di comunicazione via radio con gli altri nodi, e una fonte di energia autonoma (seppur limitata). I nodi sono programmabili: in ogni nodo sono caricati un sistema operativo e una applicazione che conferisce al dispositivo le funzionalità richieste.

1.1 Reti di sensori senza fili

La caratteristica principale delle reti di sensori risiede nel fatto che i nodi comunicano fra loro in modalità *wireless* senza infrastruttura (*ad-hoc*), creando una rete capace di organizzarsi autonomamente. La comunicazione fra i nodi serve a rendere disponibili le informazioni rilevate ad un dispositivo centrale, detto *sink*, a fare interagire i nodi per eseguire attività coordinate e per permettere al nodo centrale di trasmettere ai nodi le istruzioni da eseguire. L'utente si interfaccia alla rete di sensori attraverso un PC che è connesso al nodo sink o direttamente da un cavo (seriale o USB), con una comunicazione wireless o anche attraverso Internet, comunicazioni satellitari, eccetera (vedi fig. 1.1).

I nodi possono essere disposti nell'ambiente in una quantità che può variare da poche unità fino a centinaia o migliaia di nodi o anche più. La loro collocazione può essere effettuata in maniera casuale (per es. lanciandoli da un aereo) oppure calibrata (per es. per monitorare un edificio). Ogni nodo è soggetto a fallimento, cioè può smettere di funzionare per esaurimento delle batterie o per guasto (generalmente i dispositivi non sono resistenti agli impatti), per cui la topologia della rete non si

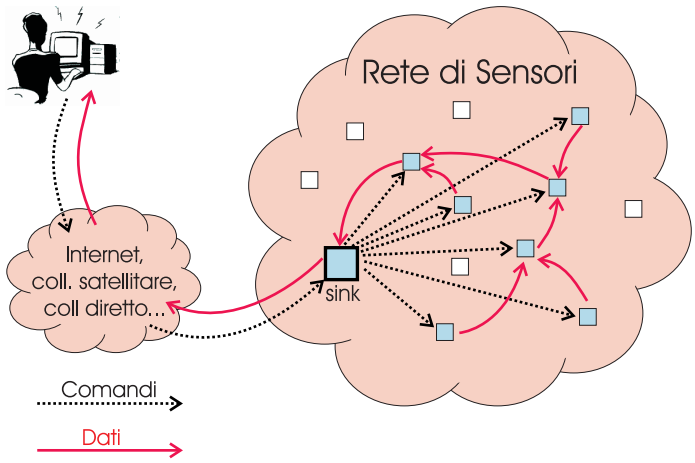


Figura 1.1: Modello di funzionamento di una rete di sensori.

mantiene identica a quella assunta nella fase di disposizione nell'ambiente, ma può cambiare continuamente. Queste considerazioni implicano che un'applicazione per una rete di sensori debba tenere conto dei seguenti vincoli di progetto:

efficienza energetica: i sensori operano alimentati da batteria, per cui è necessario che l'applicazione cerchi di mantenere il consumo energetico più basso possibile, massimizzando quindi il tempo di vita dell'intero sistema (quando un certo numero di nodi perde la sua operatività la rete non può fornire più i dati richiesti o si ha un decadimento delle prestazioni);

tolleranza ai guasti: il sistema deve funzionare correttamente anche quando alcuni nodi falliscono;

scalabilità: la rete deve poter funzionare anche con un numero di sensori molto elevato.

I nodi sensori in genere comunicano fra loro via radio (anche se sono possibili altri mezzi come gli infrarossi) in modalità ad-hoc, cioè senza un nodo centrale che crei una infrastruttura. La necessità di mantenere un consumo energetico ridotto fa sì che la potenza trasmissiva utilizzata sia piuttosto bassa, per cui l'area coperta dal segnale trasmesso risulta essere limitata e quindi un nodo può comunicare solo con i nodi più vicini. Questo aspetto comporta alcune difficoltà nel momento in cui le informazioni devono raggiungere parti della rete più lontane. In questo caso deve essere prevista la possibilità di comunicare in maniera multi-hop, cioè sfruttando alcuni nodi intermedi per l'inoltro dei messaggi verso la destinazione finale. La comunicazione multi-hop richiede l'impiego di un protocollo di routing che permetta ad ogni nodo sensore di sapere a quale vicino inoltrare un eventuale messaggio ricevuto di cui non sia il destinatario finale.

1.1.1 Esempi di utilizzo

Le possibilità di applicazione di una rete di sensori senza fili sono molteplici e spaziano da quelle ambientali, mediche, industriali, agricole, domotiche, commerciali

e militari. Ad esempio monitoraggio del traffico in una strada, individuazione dei posti liberi in un parcheggio, controllo di bambini in aree metropolitane, infrastrutture di sicurezza negli aeroporti, monitoraggio pianta per pianta in un sistema di agricoltura di precisione, gestione di aree disastrate, monitoraggio di habitat nelle riserve naturali [11, 37, 12].

Le reti di sensori possono migliorare il monitoraggio degli habitat rispetto alle tecniche tradizionali grazie alla capacità di adattarsi all'ambiente e alla loro poca invasività. Quest'ultimo punto è importante soprattutto perché consente le misurazioni senza creare disturbi. Ad esempio nell'isola di Great Duck (Maine) una rete di sensori senza fili è stata disposta per monitorare i nidi sotterranei di una specie di uccelli in via di estinzione (gli Storm Petrels di Leach) [37]. Vengono rilevati temperatura, umidità, pressione e presenza di uccelli. I dati della rete sono utili ai biologi per indagare quali sono le condizioni ideali della specie.

L'uso di un numero elevato di sensori comporta inoltre una maggiore accuratezza delle misure e le ridotte dimensioni consentono il posizionamento dei nodi anche in zone poco accessibili dall'uomo. Tra le applicazioni in questo campo troviamo la prevenzione degli incendi, l'agricoltura di precisione, lo studio di specie animali e il controllo del'inquinamento.

Una rete di sensori in un palazzo, un ponte, un aereo o nave può permettere agli ingegneri civili di valutarne l'integrità strutturale dopo un terremoto o un incendio [7]. Le misure possono essere effettuate con una risoluzione elevata grazie all'impiego di molti nodi.

Tra le applicazioni mediche troviamo il monitoraggio continuo dei pazienti in ospedale o a casa, in sostituzione agli ingombranti strumenti di misurazione fissi. Questo consente la raccolta continua dei dati clinici riducendo al minimo la degenza in ospedale. Le grandezze misurate possono poi essere memorizzate in un archivio personale, confrontate e comunicate al personale medico che può intervenire velocemente in caso di emergenza.

Le applicazioni domotiche consistono nell'integrazione dei sensori negli elettrodomestici, la cui gestione può essere automatizzata e gestita anche in remoto, ad esempio attraverso il collegamento ad Internet o tramite il telefono cellulare. La comunicazione fra i sensori consente un controllo completo della casa.

Tra le applicazioni commerciali ci sono il rilevamento della posizione e del movimento dei veicoli, il controllo del traffico, il rilevamento dei furti d'auto, la possibilità di interazione con gli oggetti in un ambiente e altri servizi di localizzazione. In un centro congressi una rete di sensori dotata di microfoni può essere utilizzata per rilevare la presenza di persone nelle stanze. In campo industriale, una rete di sensori disposta in un magazzino può essere usata per monitorare lo stato del magazzino e, posizionando nodi anche negli imballaggi, il percorso dei vari prodotti.

Ci sono anche possibilità di applicazioni militari. Ad esempio una rete di sensori disposta in un campo può riportare passaggi di truppe nemiche in tempo reale.

In generale la possibilità di "dare occhi e orecchie" all'ambiente su larga scala ha le potenzialità per cambiare radicalmente molti campi e persino la vita quotidiana.

Le applicazioni sopra elencate hanno tutti in comune la caratteristica che l'utente della rete di sensori (biologi, addetti alla sicurezza, coltivatori, ecc.) non è un'esperto di reti o di programmazione quindi una rete di sensori deve essere di facile accesso anche per un personale non specializzato.

1.2 Presentazione del problema

Le rete di sensori senza fili rappresentano una tecnologia recente e in forte sviluppo e il loro uso richiede di affrontare alcuni problemi.

La programmazione del comportamento dei nodi della rete è un compito difficile. Gli strumenti a disposizione sono ancora limitati; fare test e debugging è reso problematico anche dalla difficolta' di interazione con la rete. I programmi distribuiti che i nodi devono eseguire per coordinare le proprie attività per conseguire determinati obiettivi sono complessi e controintuitivi da progettare.

Da un punto di vista pratico, è molto oneroso dover intervenire fisicamente sui nodi di una rete già dislocata in un ambiente (soprattutto se ostile o di difficile accesso) per modificarne il comportamento e adattarlo alle esigenze che possono essere sorte dopo il suo posizionamento.

Di recente è stato proposto di far utilizzare la rete di sensori da parte degli utenti come se fosse una base di dati [29, 40, 31, 41, 34]. Questo approccio ha molti vantaggi ma le soluzioni attualmente esistenti hanno delle limitazioni.

L'intera rete dei sensori è modellata come una singola tabella (una riga per nodo). Questo pone alcuni problemi, come l'impossibilità di definire query che aggregano misurazioni effettuate in tempi diversi (per es. temperatura media nel corso di un'ora). Inoltre essendo la tabella distribuita non è possibile mettere in relazione dati provenienti da nodi diversi (per es. paragonare la temperatura di due stanze diverse).

In molti approcci correnti la stessa query viene eseguita su tutti i nodi, piuttosto che eseguire una sola query in maniera fortemente distribuita. Questo significa che l'ottimizzazione è fatta su considerazioni globali e non è possibile ottimizzare query per una specifica configurazione dei nodi. Inoltre non viene considerato il caso di reti in cui i nodi hanno insiemi diversi di transduttori.

Infine, l'ottimizzazione di query per il caso delle reti di sensori ha delle particolarità che non sono presenti nel caso dei generali database distribuiti. Le tecniche standard di ottimizzazione sono solo parzialmente utili nel nostro caso: ne servono di nuove specificamente progettate per questo caso.

1.3 Contenuto e contributi della tesi

Il lavoro di questa tesi si basa sul progetto MaD-WiSe, che è finalizzato alla progettazione di un sistema per gestire interattivamente una rete di sensori senza fili.

A questo fine la tesi definisce MW-SQL, un nuovo linguaggio per interrogare una rete di sensori senza fili similmente a come si farebbe per un database tradizionale. Viene anche definito un ottimizzatore di query per reti di sensori. La particolarità della strategia di ottimizzazione è dovuta al fatto che si cerca di minimizzare il consumo di energia per prolungare la durata delle batterie nella rete di sensori. Il tutto è stato realizzato ed integrato nel sistema MaD-WiSe.

La tesi è strutturata come segue:

- il capitolo 2 descrive l'hardware utilizzato e analizza brevemente lo stato dell'arte nel campo delle reti di sensori;
- il capitolo 3 mostra il sistema MaD-WiSe, compreso il modello dei dati e l'algebra su stream usati;

- nel capitolo 4 viene definito il linguaggio di interrogazione MW-SQL;
- il capitolo 5 descrive il processo di traduzione di una query in MW-SQL in un piano di esecuzione da disseminare ed eseguire nella rete;
- il capitolo 6 descrive l'ottimizzatore delle query;
- il capitolo 7 mostra dei dettagli relativi all'implementazione e riporta alcuni esempi dei risultati ottenuti;
- infine nel capitolo 8 si traggono alcune conclusioni e si delineano alcuni possibili direzioni di sviluppo futuro.

Capitolo 2

Stato dell'arte e tecnologie esistenti

In questo capitolo verrà brevemente passato in rassegna lo stato dell'arte relativo all'hardware e software dei nodi della rete, ai sistemi per gestire le reti di sensori senza fili e infine alle tecniche di ottimizzazione per le query.

2.1 Descrizione dei nodi

In generale un nodo della rete di sensori è un piccolo dispositivo composto da:

- transduttori: dispositivi analogico-digitali che rilevano dati dell'ambiente come temperatura, umidità, pressione atmosferica, accelerazione (propria), suono, campo magnetico, luce, o altro. I transduttori sono modularmente montati sul pannello dei sensori del nodo. Di solito i nodi di una rete hanno tutti gli stessi transduttori ma questo non è necessariamente il caso. Comunque i transduttori montati dipendono dall'applicazione. Ogni modello di nodo ha un numero massimo di transduttori che può alloggiare;

- antenna: ricetrasmittente, permette ad un nodo di mandare e ricevere (ma non contemporaneamente) messaggi a quelli vicini usando onde radio. Il raggio di trasmissione è molto piccolo, da qualche metro a poche decine di metri. La comunicazione è broadcast: tutti i nodi entro il raggio ricevono fisicamente il messaggio. Siccome la frequenza radio è la stessa per tutta la rete, ci possono essere interferenze;

- processore: i motes hanno una capacità limitata di calcolo, grazie ad una CPU montata su un singolo chip. Possono eseguire programmi generici che hanno il controllo sugli altri dispositivi. Chiaramente questi programmi utilizzano una memoria temporanea. I nodi sono, anche, dotati di memoria permanente su supporto *EEPROM*;

- sistema di stand-by e timer: un nodo si può accendere o spegnere automaticamente usando un timer. O può accendersi come risultato di un movimento o altro registrato da un transduttore;

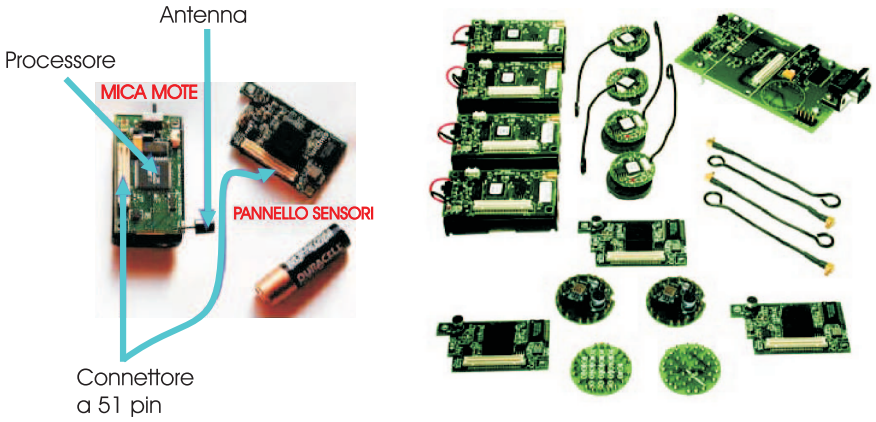


Figura 2.1: Un modello di nodo Mica-z e un set di modelli Mica-2.

Caratteristiche	Mica-2	Mica-z
Modello CPU	ATMEGA 128	ATMEGA 128
Freq. di clock CPU	8 MHz	8 MHz
Memoria RAM	4 KBytes	4 KBytes
Memoria Flash	128 KBytes	128 KBytes
Modello Mem. Flash	Atmel AT45DB41	Atmel AT45DB41
Freq. Radio	76 Kbps	250 Kbps

Tabella 2.1: Caratteristiche hardware dei modelli Mica-2 e Mica-z.

- batteria: ogni nodo ha una fonte di energia propria anche se limitata, che viene utilizzata per alimentare i sensori, il processore, il timer e l'antenna (ogni dispositivo ha un consumo diverso). Come già detto, questo rappresenta il limite più grande delle reti di sensori perché cambiare le batterie è poco pratico. Molto del lavoro fatto sulle reti di sensori serve per far allungare l'autonomia della rete stessa;

- protezione esterna: siccome i nodi in molte applicazioni sono esposti agli elementi ambientali devono essere adeguatamente protetti.

2.1.1 Modelli

Anche se tutti i nodi condividono, a grandi linee, la descrizione sopra indicata, ogni specifico modello avrà le sue caratteristiche (vedi [19]). In pochi anni si è passati da mote sviluppati dall'università per la ricerca (Berkley-motes) a prodotti commerciali (per esempio Amtel Motes della Amtel e i-Motes della Intel).

Il lavoro quì presentato non presuppone l'uso di uno specifico modello. Tuttavia il nostro progetto è stato testato su due modelli diversi: Mica-2 e Mica-z (fig. 2.1). Questi modelli, attualmente in commercio [24], hanno le caratteristiche mostrate nella tabella 2.1.

2.1.2 TinyOS

Dal punto di vista del software, su questi dispositivi gira un sistema operativo apposito.

TinyOS [20] è un sistema operativo open source, sviluppato dalla University of California di Berkeley e progettato per sistemi *embedded*, in particolare nellambito delle reti di sensori. Data la possibilità di modificare il codice, questo sistema operativo è diventato la piattaforma di sviluppo per ogni soluzione proposta nel campo delle reti di sensori.

TinyOS è composto da una serie di librerie che realizzano componenti di alto livello e da moduli che consentono di accedere all'hardware senza che il programmatore ne conosca i dettagli. Ad un livello più basso si trovano invece le parti che si occupano di gestire il bootstrap del dispositivo, la schedulazione delle operazioni, il power management, le interfacce di comunicazione, i sensori montati sulla board e le eventuali periferiche esterne.

Inoltre, il sistema operativo si occupa della sincronizzazione degli orologi dei nodi con quelli dei vicini e delle comunicazioni.

In particolare il modulo *MAC* (Medium Access Control) di TinyOS si occupa di procurare un livello astratto di comunicazione uno a uno anche se fisicamente la comunicazione radio è, naturalmente, uno a molti (broadcast). Questo è ottenuto anteponendo l'ID del destinatario ad ogni messaggio e ignorando i messaggi avvertiti ma non destinati al nodo in questione. Tuttavia il sistema operativo permette di fare *snooping*, cioè interpretare i messaggi rilevati dalla propria antenna ma aventi un destinatario diverso da sé (ciò ha un costo, perché bisogna tenere accesa l'antenna sino alla fine del messaggio, ma può essere utile per qualche ottimizzazione — vedi par. 2.3.2). Il livello MAC si occupa anche della gestione delle interferenze tra le comunicazioni radio di nodi diversi e di mandare messaggi di *acknlowdgement*.

Le applicazioni basate su TinyOS, così come il sistema operativo stesso, sono scritte utilizzando il linguaggio *NesC* che è un'estensione del linguaggio C per programmare i sistemi embedded.

I motes Mica2 e Micaz sono compatibili con il sistema operativo TinyOS.

2.2 Sistemi per gestire wireless sensor network

Anche se le reti di sensori senza fili sono un campo di ricerca molto recente, sono già stati proposti da vari gruppi di ricerca vari approcci.

2.2.1 Approccio sviluppato dalla Siemens

Alla Siemens stanno sviluppando metodologie che hanno strette relazioni con le reti di sensori [34]. Nel contesto delle applicazioni di logistica postale, propongono l'uso di una vasta rete (highly distributed system) i cui nodi siano piccoli palmari, ciascuno a bordo dei corrieri che trasportano fisicamente gli oggetti, fra loro interconnessi da comunicazioni radio, che gestiscono in tempo reale le informazioni di tutti gli oggetti trasportati, delle loro destinazioni, la posizione corrente del corriere (attraverso sensori GPS), eccetera.

Nonostante le differenze, alcuni problemi affrontati in questo progetto, anche secondo gli stessi autori, sono molto simili a quelli riscontrati in una rete di sensori: la rete è ad-hoc e deve connettere un numero molto elevato nodi (scalabilità); i dati raccolti hanno breve durata (ad es. quelli di un'ora fa probabilmente non sono più validi); il software che deve essere leggero perché deve essere eseguito su hardware dalle caratteristiche molto limitate (anche se in misura minore che non nel caso delle reti di sensori); il sistema deve essere affidabile e prevedere che ogni nodo potrebbe smettere di funzionare o essere incapace di comunicare in qualsiasi momento.

Anche se sono previsti in futuro reti di circa centinaia di migliaia di nodi, i lavori pubblicati riportano per ora simulazioni di soli centinaia di nodi. L'approccio sviluppato alla Siemens si basa principalmente su alcune idee:

- affrontare il problema di routing con protocolli simili a quelli usati nel protocollo IP di internet, cioè un protocollo di routing non centralizzato e dinamico: i router hanno una conoscenza precisa dei nodi intorno e vaga delle parti di rete lontane e questa conoscenza si aggiorna col tempo;

- non usare un nodo base che "sappia tutto" (gli indici del database, i metadati, la struttura della rete, ecc.), perché ciò andrebbe contro la affidabilità del sistema: se fallisce il nodo centrale (front-end) non funziona più la rete.

Rimane da vedere se questo approccio, che pone l'accento sulla decentralizzazione, sia applicabile con profitto al caso di rete di sensori vere e proprie. Quello che non sembra considerato è che in una rete di sensori il nodo front-end (base-station) è molto diverso dagli altri nodi (ad es. MICA Motes): ha imparagonabilmente maggiore memoria e potenza di calcolo, energia illimitata, una robustezza maggiore perché è protetto dalle condizioni ambientali. Quindi nel nostro caso ha senso avere una struttura di rete più centralizzata.

Tuttavia in questo approccio concordano su un importante concetto che è applicabile direttamente anche al nostro caso: dal punto di vista dell'utente finale della rete l'interfaccia del sistema può essere convenientemente resa simile ad SQL, adattato e potenziato con appositi costrutti, e la rete stessa va considerata come una specie di database.

2.2.2 Approccio Fjord

Fjord [30] è un'architettura per gestire query sulle reti di sensori. Ha due punti di forza: (a) permette agli utenti della rete di sensori di fare query per mettere insieme data stream provenienti da sensori e tabelle di data base tradizionali salvate su disco; queste query utilizzeranno dunque operatori che sono misti sia *pull-based* che *push-based*; (b) introduce degli operatori, detti sensor proxy, che fanno da mediatori tra un query processor centralizzato e i sensori fisici finalizzati al risparmio energetico. Ogni proxy si occupa di un sottoinsieme della rete di sensori ed è connesso al server con un cavo internet o un collegamento radio.

Il sistema segue questo schema: gli utenti immettono una query al sistema che la processa e la suddivide tra i proxy necessari alla query; i proxy eseguono le query assegnateli comunicando con i nodi e mandano i dati risultanti al server; quando ci sono moti utenti che utilizzano la rete di sensori e quindi molte query

concorrenti il proxy riutilizza gli stessi dati per tutte le query che li richiedono. Il proxy ha anche altre funzioni come gestire il passo di campionamento a seconda delle richieste degli utenti, spegnere e riaccendere i nodi in funzione della loro utilità e in generale determinare il comportamento dei nodi (ad esempio in quali casi mandare le misurazioni).

2.2.3 Approccio Cougar

L'approccio Cougar [40, 41, 9] consiste in un'architettura per la gestione delle reti di sensori a grandi linee simile a quella che adottiamo noi. Anche quì un nodo centrale (gateway) ha a disposizione molte informazioni sullo stato attuale della rete (un *catalogo*) e utilizza queste informazioni per ottimizzare piani di esecuzione di query da disseminare nei nodi della rete opportuni per essere eseguito.

In questo approccio il modello dei dati prevede sia relazioni memorizzate sul disco che sequenze di misurazioni. Queste ultime sono modellate attraverso un Abstract Data Type (ADT), gestito su un apposito "proxy" eseguito su ogni nodo. L'ADT relativo ad un sensore metterà a disposizione funzioni di *signal processing* fra cui, ad esempio, quelle che restituiscono una misurazione scalare (ad es. getTemp in un ADT relativo ad un sensore di temperatura). La sequenza di misurazioni è ottenuta con esecuzioni successive di queste funzioni.

La gestione delle query in questo approccio è basato su tre livelli:

- il primo livello è eseguito sui nodi. I nodi hanno un motore di esecuzione delle query che esegue le funzioni di signal processing incluse nella corrispondente interfaccia ADT. I nodi sono raggruppati in *cluster* e ad ogni cluster si assegna un nodo *leader*;

- il secondo livello è composto dai nodi leader dei cluster che coordinano ed eseguono gli operatori di aggregazione dei dati prodotti nei nodi del rispettivo cluster. I leader ricevono i dati risultanti dalle funzioni di signal processing eseguiti nei nodi e anche i dati già parzialmente aggregati dai nodi. I leader eseguono l'aggregazione finale e mandano il risultato al front-end;

- il terzo livello è eseguito nel front-end, che elabora i dati ricevuti (accedendo anche alle tabelle memorizzate) e li mostra all'utente.

Gli autori di questo approccio fanno notare che determinare la quantità e il tipo di informazione (la posizione dei sensori, la loro connettività, la stabilità della rete, la batteria residua, l'affidabilità delle comunicazioni, metadati relativi ai dati acquisiti, distribuzione di probabilità) da tenere nel catalogo è un importante problema di ricerca aperto: maggiori informazioni consentono migliori ottimizzazioni ma hanno un costo maggiore per essere tenute aggiornate. Nel nostro sistema, come vedremo, vengono tenute poche informazioni: la posizione dei sensori, il numero e il tipo di transduttori presenti in un sensore e la dimensione in byte delle misurazioni. In futuro potrebbero essere aggiunti e mantenuti aggiornati altri metadati che sarebbero utili, ad esempio, per calcolare i fattori di selettività.

2.2.4 Approccio TAG

TAG (Tiny Aggregation service) [31] si concentra sul problema della definizione, esecuzione, ottimizzazione degli aggregati in una rete di sensori senza fili.

L'idea centrale è di calcolare gli aggregati in-network, cioè all'interno della rete man mano che i dati vengono raccolti e convogliati verso il nodo gateway.

TAG permette la definizione di nuovi aggregati. Un aggregato è definito da 3 operazioni: inizializzazione (di uno stato parziale), merge (di due stati parziali) e valutazione finale (da stato parziale a risultato finale). Questo fornisce una grande flessibilità nella definizione di nuovi aggregati.

Inoltre, gli aggregati sono classificati a seconda delle loro caratteristiche: ad esempio si distinguono quelli *duplicate insensitive* da quelli *duplicate sensitive* perché nei primi il risultato non cambia se si aggiungono delle copie degli operandi e nei secondi si (quindi l'operatore *max* sarà duplicate insensitive mentre il Sum sarà sensitive).

Le caratteristiche degli aggregati sono metadati che servono a decidere quale ottimizzazioni usare nel loro calcolo. Infatti l'approccio TAG introduce alcune strategie di ottimizzazione specifiche per il calcolo degli aggregati in una rete di sensori (vedi par. 2.3.2 sotto). Le ottimizzazioni sono introdotte automaticamente in modo invisibile all'utente.

Il sistema per calcolare gli aggregati in-network di TAG prevede la possibilità di raggruppare i nodi in sottoinsiemi similmente al costrutto Group by di SQL. Questo significa che ogni nodo deve tenere traccia di uno stato parziale per ogni gruppo a cui appartiene, senza farne il merge. Se la memoria non è sufficiente si manda uno stato parziale relativo a un gruppo su al padre nell'albero di routing e se è necessario così via fino alla radice.

Il sistema TAG prevede che durante l'esecuzione della query in ogni dato istante solo alcuni dei nodi siano attivi mentre altri siano in modalità stand-by grazie ad una sincronizzazione nella fase di disseminazione di query (ogni nodo, nel propagare il piano di esecuzione della query, specifica l'intervallo di tempo nel quale si aspetta di ricevere i risultati parziali).

Le giunzioni di dati non aggregano diversi dati in uno solo e quindi non sono gestiti nell'approcccio TAG.

2.2.5 Approccio TinyDB

TinyDB [29, 17] è un processore di query "acquisizionale" distribuito che viene eseguito sul processore di ogni nodo. Si chiama acquisizionale perché i dati gestiti sono acquisiti attraverso i transduttori piuttosto che letti dal disco. TinyDB è progettato per i Mica mote di Berkeley ed eseguono sulla piattaforma TinyOS (vedi par. 2.1.2). Sfruttano, come nel nostro caso, la capacità dei sensori di controllare quando e come i dati vadano acquisiti, e fornisce elementi comuni del query processor come proiezioni, restrizioni, giunzioni e aggregati.

TinyDB vede la rete di sensori come un'unica tabella "Sensors" con una riga per ogni misurazione di ogni mote e una colonna per ogni quantità misurata da un sensore (temperatura, pressione, ecc.). In più avremo delle colonne come "NodeId" che contiene l'identificatore del mote che ha effettuato la misurazione, e "Timestam-

p" che contiene l'ora in cui è stata fatta la misurazione, in millisecondi (ottenuta leggendo il timer).

Similmente al nostro caso l'utente immette query (long running) nel sistema che dissemina un piano di esecuzione risultante fra i nodi appropriati. Ogni nodo esegue al suo interno la stessa sequenza di acquisizioni e operazioni. I dati così acquisiti da ogni nodo seguono il percorso inverso e vengono mostrati all'utente.

Un aspetto importante di TinyDB consiste nelle query basate su eventi, cioè query che vengono istanziate ed eseguite ogni volta che si verifica un certo "evento". è possibile che in un determinato momento molte istanze di query di questo tipo siano attivate contemporaneamente. Questo pone il problema che ciascuna istanza consuma le risorse, soprattutto la batteria dei Mote. Per ridurre questo consumo si usa una tecnica di ottimizzazione multi-query che si basa su descrivere queste query in modo che tutte le istanze riutilizzino delle sottoparti dell'unica tabella sensori.

In TinyDB è possibile anche formulare una query specificandone la durata della sua esecuzione, e il sistema calcolerà le frequenze di campionamento tenendo conto dell'energia residua nelle batteria e dei consumi preventivati.

Questo approccio ha dei limiti:

- visto che tutti i nodi hanno lo stesso piano, non è possibile fare ottimizzazioni globali, che riservino ad ogni nodo un ruolo diverso;
- per lo stesso motivo, non è possibile mettere in relazione dati provenienti da nodi diversi (ad esempio, richiedere se la temperatura di una stanza sia maggiore di quella di un'altra);
- non c'è un meccanismo che consenta di eseguire aggregati temporali, cioè aggregare insieme dati misurati da uno stesso transduttore in momenti diversi;

2.2.6 Approccio CAPS

Collective Adaptive Precision Setting (CAPS) è un sistema sviluppato dai centri di ricerca IBM per la gestione di rete di sensori [23]; si basa su una nuova categoria di ottimizzazione delle query che sacrifica la precisione assoluta dei risultati per ottenere un grande risparmio del consumo di energia.

Si specifica un intervallo di tolleranza e il sistema garantisce che i risultati della query non differiranno mai dalla realtà di più di questa tolleranza.

Il principio è quello della predizione dei risultati. In pratica sia il nodo che acquisisce i dati che il computer centrale hanno a disposizione un "predittore" che calcola il risultato ritenuto più probabile per un certo valore da misurare. Se il nodo della rete dopo aver acquisito il dato si accorge che la misura effettiva si discosta di troppo poco dal valore previsto, allora si risparmia di trasmettere il dato risparmiando il costo di trasmissione. Questo silenzio verrà interpretato come la conferma della predizione.

I predittori potranno utilizzare dati statistici, o metadati. L'importante è che questi dati siano a disposizione sia del computer centrale che del nodo che acquisisce; infatti la predizione dovrà risultare identica affinché il sistema funzioni.

Questo approccio richiede di definire buoni predittori. Nel caso di calcolo degli aggregati, bisogna anche calcolare l'effetto di un errore negli operandi sul risultato.

2.3 Ottimizzazioni di query

Un contributo importante del lavoro quì presentato (vedi cap. 6) consiste nella definizione e realizzazione di un ottimizzatore per le query definite nel linguaggio introdotto che sfrutta le particolarità del nostro sistema. L'ottimizzazione di query è un campo molto studiato sia nel contesto dei tradizionali database (sia centalizzati che distribuiti) che più recentemente nel contesto delle rete di sensori senza fili.

2.3.1 Ottimizzazione in Databases

Il caso dei database tradizionali è simile per certi aspetti al caso della rete di sensori. Data una query dell'utente il sistema di gestione del database (DBMS) centralizzato può tradurla in molti piani di esecuzione diversi [2, 26]. Questi piani sono equivalenti per quanto riguarda il risultato che producono ma possono presentare costi molto diversi. Il compito dell'ottimizzatore è di scegliere il piano meno caro. Come nel nosrtro caso una query segue diversi passi per essere eseguita:

- il parser analizza la query e verifica la sua correttezza; se corretta viene tradotta in una rappresentazione interna del piano;
- l'ottimizzatore genera piani equivalenti e sceglie quale l'implementazione usare per ogni operatore; per ogni piano viene valutato un costo, il piano di costo minore viene scelto;
- il generatore di codice traduce il piano ottimizzato in una serie di chiamate al processore di query;
- il processore di query esegue queste chiamate.

In particolare il caso dei database distribuiti [10] ricorda ancora più da vicino il nostro caso, perché il piano di esecuzione deve anche decidere l'allocazione di esecuzione di ogni operatore e inoltre un costo molto importante da minimizzare è quello di trasmissione dei dati.

Tuttavia una differenza fondamentale è che nel nostro caso non abbiamo tabelle fisse di dimensione finita memorizzate su un disco ma stream di dati da processare ennupla dopo ennupla al volo. Inoltre il piano deve anche descrivere come effettuare il campionamento dai sensori e bisogna anche tener conto del limite delle batterie.

Un'altra differenza sta nel funzionamento degli operatori: mentre nel DBMS tradizionale o distribuito gli operatori sono pull-based, cioè un operatore richiede dati dai suoi operandi nel piano di esecuzione solo quando ne ha bisogno; invece nel caso di rete di sensori senza fili gli operatori sono push-based, cioè gli operandi mandano autonomamente le ennuple all'operatore, che può ignorarle o accetterle ma in quest'ultimo caso deve processarle al volo.

Queste differenze comportano diversità nel linguaggio in cui si esprime la query, diversità negli operatori e nelle loro implementazioni e diversità nell'ottimizzazioni applicate.

2.3.2 Ottimizzazione sulle reti di sensori

Le tradizionali metriche di costo (per es. quanti accessi al disco sono necessari per recuperare i dati) non si applicano alle query su data stream non limitate. Metriche di costo maggiormente appropriate sono:

- accuratezza [23] e ritardo nei risultati;
- frequenza in uscita [38];
- consumo di energia [29, 41].

In particolare l'ultima metrica è consigliabile perché la capacità di batteria di solito è scarsa: minimizzare il consumo di energia porta il grande vantaggio di estendere l'autonomia della rete rimandando l'intervento umano per ricaricare o sostituire le pile; si noti che la rete di sensori potrebbe essere estesa per monitorare ambienti rocciosi o difficili da raggiungere.

Come abbiamo accennato, nelle reti di sensori gli operatori, di solito, sono push-based. Alcuni sistemi [30, 6], tuttavia, utilizzano anche operatori parzialmente o totalmente pull-based. Anche nel sistema MaD-WiSe, su cui si basa questo lavoro, l'ottimizzatore si avvantaggierà di uno speciale operatore misto descritto nel paragrafo 3.5.1.

Come nel DBMS tradizionale, un importante ottimizzazione consiste nello scegliere il giusto ordine degli operatori join, perché ciò può cambiare sensibilmente il costo di un piano. Alcuni sistemi [32, 35, 13] cambiano dinamicamente il piano durante l'esecuzione per riflettere le condizioni del sistema; questo è ottenuto con politiche di routing che tentano di individuare gli operatori da eseguire per primi perché sono veloci e selettivi. Nel sistema che proponiamo, un'apposita pre-ottimizzazione (vedi par. 6.1) riordina e rialloca gli operatori binari per minimizzare la distanza dei nodi che comunicano fra di loro.

I data stream possono arrivare da altri nodi. Una strategia di ottimizzazione consiste nel ridurre la comunicazione riordinando gli operatori tra i nodi [14] e eseguendo alcune funzioni come filtraggio, aggregazioni e proiezioni localmente nei nodi [41, 31, 33]. Anche il nostro sistema utilizza tecniche di questo tipo di calcolo nella rete stessa degli operatori e l'ottimizzatore (vedi par. 6.2) cerca di trarne il massimo vantaggio.

Sono state anche proposte altre specifiche tecniche di ottimizzazione per reti di sensori [4, 8, 39, 35]. Per esempio in [31] si sfrutta il fatto che i nodi comunicano a basso livello in modalità broadcast. Se una query contiene un aggregato `min`, un nodo non propaga il valore da lui rilevato se ha ascoltato da uno dei suoi vicini un messaggio che riporta un valore misurato più piccolo del suo. Tale messaggio non necessariamente è indirizzato al nodo stesso, bensì può anche essere ascoltato in modalità snooping. Inoltre un nodo potrebbe mandare copie ridondanti del minimo valore a lui noto a tutti i suoi vicini al fine di minimizzare le possibilità di perdere i messaggi. Questa tecnica è inapplicabile ad altri aggregati come `sum` e `avg` perché le copie replicate altererebbero il risultato.

Anche l'ottimizzazione basata sui predittori descritta nel paragrafo 2.2.6 è un esempio di tecnica tecniche di ottimizzazione specifica per reti di sensori.

Un'altro modo per ottimizzare [42] consiste nello schedulare accuratamente le comunicazioni in modo da evitare che si verifichino interferenze fra messaggi, piuttosto che aspettare il livello di comunicazione MAC le rilevi e le risolva costosamente.

Un problema a parte è l'ottimizzazione nei casi in cui molte query simili vengono eseguite contemporaneamente sulla stessa rete. Un tipo di soluzione [13, 32] consiste nel costruire una tabella delle query attive e, ogni volta che arriva una ennupla contenente una misurazione, verificare per ogni query attiva se l'ennupla arrivata deve far parte dei risultati di quella query.

Capitolo 3

L'approccio MaD-WiSe: architettura, modello dei dati e algebra su stream

Si descrive il sistema MaD-WiSe [5] (Management of Data in Wireless Sensor networks) che è un progetto congiunto del laboratorio Wireless Networks and Multimedia Networked Information System dell'istituto ISTI del CNR di Pisa e del dipartimento di Informatica dell'Università di Pisa. Lo scopo del progetto è l'integrazione di tecnologie di database con reti di sensori senza fili. Mostremo il modello dei dati adottato dal sistema sul quale ci siamo basati (che era già stato definito precedentemente) e l'algebra degli operatori usati per le query, la cui definizione è uno dei contributi di questo lavoro.

L'implementazione relativa a questo lavoro e descritta nei prossimi capitoli integra la versione precedente del sistema MaD-WiSe. Il sistema stesso non solo è stato ampliato con nuove funzionalità ma è stato anche adattato alle nuove esigenze.

3.1 Obiettivi

Il sistema MaD-WiSe permette di pilotare una rete di sensori determinando il comportamento dei nodi che la compongono i quali guidati dal sistema, raccolgono dati attraverso il rilevamento dei transduttori, li processano e li trasmettono. Per questo scopo MaD-WiSe utilizza un modello di dati (vedi par. 3.4) basato su stream a granularità molto fine: i dati prodotti da un singolo transduttore vengono modellati come un singolo stream di dati.

L'algebra del processore di dati (vedi par. 3.5) è composta da operatori che prendono in input stream di dati e producono in output stream di dati.

Un piano di esecuzione della query è rappresentato da una combinazione di operatori dell'algebra connessi da stream. Gli stream di dati possono connettere operatori eseguiti su nodi differenti offrendo la possibilità di distribuire l'elaborazione dei dati nella rete.

Uno dei vantaggi dell'approccio MaD-WiSe consiste nella possibilità di collocare fisicamente la rete in un territorio e in seguito di definire, correggere o modificare le applicazioni che usano la rete senza più intervenire fisicamente nella rete stessa.

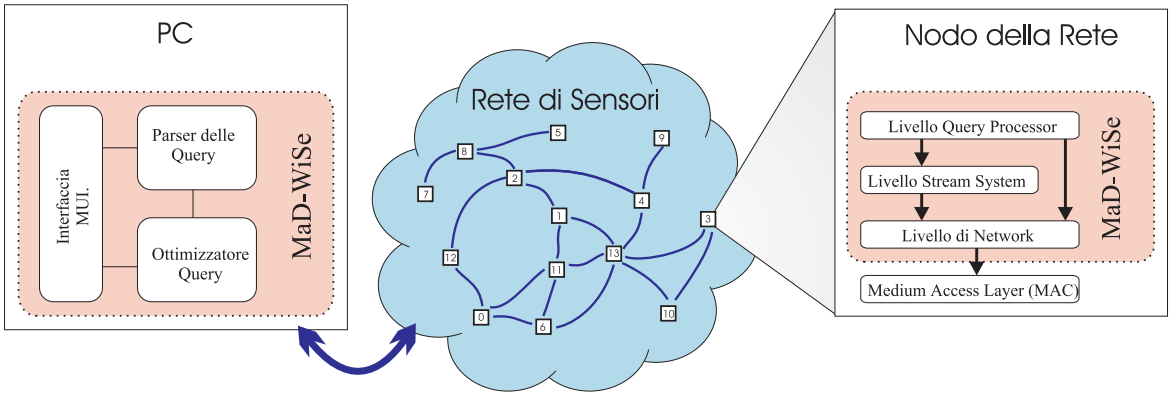


Figura 3.1: L'architettura del sistema MaD-WiSe.

3.2 Architettura del sistema MaD-WiSe

Lo stato di applicazione del sistema MaD-WiSe è suddiviso in due parti principali (vedi fig. 3.1):

- la prima viene eseguita sul processore di ogni nodo della rete ed è responsabile di implementare l'effettiva acquisizione dei dati, la loro elaborazione all'interno della rete e l'instradamento verso il nodo sink;
- la seconda viene eseguita su un PC connesso al nodo sink.

La prima parte è suddivisa in tre livelli (vedi fig. 3.1, a destra):

livello di Network: il compito di questo livello è fornire servizi di comunicazione sulla rete di sensori, gestendo la comunicazione fra coppie di nodi: quando i livelli superiori richiedono che due nodi debbano comunicare, il livello di rete organizza questa comunicazione sincronizzando i due nodi e specificando eventuali nodi intermedi che devono prendere parte alla comunicazione secondo il modello multi-hop. Questo livello si appoggia sul sottostante livello Medium Access Control (MAC) fornito dal sistema operativo TinyOS eseguito sui nodi. Questo a sua volta fornisce la comunicazione uno a uno tra nodi e si occupa di ripetere la trasmissione del messaggio in caso di interferenze, cattiva recezione o altro;

livello Stream System questo livello si occupa dello stato di trasporto dei dati gestendo gli stream di dati prodotti dai nodi (vedi par. 3.4);

livello Query Processor questo livello consiste in una serie di moduli che implementano gli operatori dell'algebra di MaD-WiSe usati per eseguire le query (vedi par. 3.5).

Questa parte di MaD-WiSe si basa sul sistema operativo TinyOS che consente di utilizzare le funzionalità base dei nodi sensori. Quando la rete viene inizializzata il sistema provvede ad assegnare delle coordinate virtuali ad ogni nodo.

La seconda parte del sistema (vedi fig. 3.1 a sinistra) comprende l'**Interfaccia Grafica Utente di MaD-WiSe (MUI)** che permette all'utente di programmare,

attraverso piani di esecuzione delle query, la rete di sensori e di vederne i risultati. Per ottenere questo il sistema utilizza un parser di query capace di generare piani di esecuzione e un ottimizzatore di query.

Rispetto alla versione precedente a questo lavoro [5], l'interfaccia MUI ha subito molti cambiamenti anche concettuali per riflettere le nuove funzionalità introdotte con questo lavoro. Nella versione precedente la specifica del piano di esecuzione avveniva creando manualmente, attraverso appositi pulsanti e finestre di dialogo di MUI, ogni singolo stream e operatore che componeva il piano. Questo processo era totalmente flessibile ma laborioso ed inoltre costruire una query complessa in maniera corretta e ottimizzata, lavorando a un livello così basso, richiedeva un'accorta e difficile opera di pianificazione.

Ora lo stesso risultato è ottenuto traducendo (cap. 5) query espresse dall'utente in un linguaggio ad alto livello (cap. 4) appositamente pensato e l'ottimizzazione è stata automatizzata (cap. 6).

Nel paragrafo successivo, mostreremo l'interfaccia della versione precedente di cui molti elementi sono comunque rimasti inalterati nell'interfaccia attuale (descritta nel cap. 7).

3.3 Interfaccia grafica utente di MaD-WiSe (MUI)

L'interfaccia grafica utente del sistema MaD-WiSe, che chiamiamo MUI, è implementata in Java ed eseguita su un PC (base-station) connesso al nodo sink. Il suo scopo è di permettere all'utente di interagire con la rete di sensori tramite query e di visualizzare i risultati.

Nella versione del sistema precedente a questo lavoro, l'unico modo di definire le query è quello di costruire manualmente un piano di esecuzione funzionante per la query che si intende eseguire. Questo viene fatto definendo uno ad uno tutti gli elementi che compongono il piano (stream sensori, stream remoti, stream locali che connettono opportuni operatori).

L'interfaccia (vedi fig. 3.2) è composta da un riquadro principale dove viene raffigurata la rete e da una barra degli strumenti posta alla sinistra del riquadro. I nodi della rete ciascuno raffigurato da un rettangolo numerato ed allocato nella posizione individuata dalle sue coordinate virtuali sono visibili nel riquadro principale. Cliccando su un nodo si sceglie quale sensore è attivo.

Nella barra degli strumenti c'è un pulsante per ogni tipo di nuovo elemento che si può definire (uno per gli stream, uno per l'operatore join, uno per l'operatore di proiezione ecc.). Il nuovo elemento creato è allocato nel sensore correntemente attivo. Il pulsante corrispondente fa aprire una finestra di dialogo in cui l'utente specifica tutti i parametri necessari per definire l'operatore o lo stream che viene creato. Ad esempio, per creare l'operatore di proiezione bisognerà specificare quali sono gli attributi proiettati, quali sono gli identificatori dello stream di ingresso e di uscita (che devono essere creati precedentemente). Invece per creare uno stream sensore bisognerà specificare nella finestra di dialogo il transduttore usato, un nuovo identificatore per lo stream, il tipo, che può essere su richiesta o periodico, e in quest'ultimo caso anche il passo di campionamento.

Il pulsante etichettato *View* permette di aprire un riquadro specifico del nodo,

cioè il riquadro di query che riporta gli operatori e gli stream creati in quel nodo fino a quel momento.

Gli operatori sono rappresentati con dei cerchi bianchi che riportano il simbolo dell'operatore. Gli stream sensori sono rappresentati da un cerchio colorato che riporta il simbolo del transduttore campionato e un segmento (dello stesso colore) che lo collega agli altri operatori e che è etichettato con l'identificatore dello stream. Ad esempio nella figura 3.2 in primo piano a destra si ha il riquadro della query del nodo 4 che mostra uno stream sensore fotometrico identificato con SS0 e dove L è il simbolo del transduttore campionato, cioè *Light*.

Gli stream locali sono raffigurati con segmenti gialli che collegano gli operatori e che sono etichettati con l'identificatore dello stream.

Gli stream remoti sono rappresentati con segmenti verdi (anche loro etichettati con l'identificatore dello stream), e sono presenti sia nel riquadro di query del nodo che trasmette sia in quello che riceve.

Posizionando il puntatore del mouse sopra i vari elementi grafici si attiva un piccolo riquadro che riporta i parametri dell'operatore o stream corrispondente.

Il pulsante della barra degli strumenti etichettato *Start* ordina all'applicazione di mandare i messaggi necessari ai nodi fisici per farli agire come definito graficamente dall'utente (cioè inizia l'esecuzione della query). In pratica la MUI opera mandando messaggi al sink che è fisicamente connesso al PC e il query processor eseguito sul nodo sink inoltra questi messaggi nella rete di sensori.

Tramite il pulsante *Stop* si ordina all'applicazione di mandare un messaggio al sink per fermare l'esecuzione della query.

Il pulsante *Data* è attivo solo quando si sta eseguendo una query e fa comparire una finestra che contiene una tabella che aumenta nel tempo di dimensione con i dati inseriti in essa così come arrivano al *Serial* del nodo sink e attraverso questa funzionalità l'utente visualizza il risultato della query.

Il paragrafo 7.2 mostrerà come questa interfaccia sia stata estesa per riflettere le funzionalità che implementano quanto presentato in questo lavoro.

3.4 Modello dei dati

Come nei database relazionali [3], un tipo ennupla è un insieme di coppie (etichetta, tipo primitivo) ed un valore di tipo ennupla è un insieme di coppie (etichetta, valore), dette campi, con le stesse etichette del tipo e con valore del corrispondente tipo primitivo. Nel sistema MaD-WiSe tutti i campi hanno lo stesso tipo, e i valori sono sempre numeri naturali.

Modelliamo i dati gestiti da una rete di sensori come stream di ennuple con lo stesso tipo e in numero non limitato superiormente a priori, cioè col passare del tempo arrivano sempre nuove ennuple. Ad esempio, un termometro elettronico attivato ogni 10 secondi produce uno stream, cioè una sequenza teoricamente illimitata di ennuple contenente ciascuna la temperatura rivelata e il tempo di rilevazione, e un'ennupla è prodotta ogni 10 secondi. Chiameremo questo tipo di stream "stream sensore" poiché contiene dati rilevati da un sensore. Invece gli "stream remoti" trasportano dati da un nodo all'altro della rete di sensori e gli "stream locali" trasportano dati

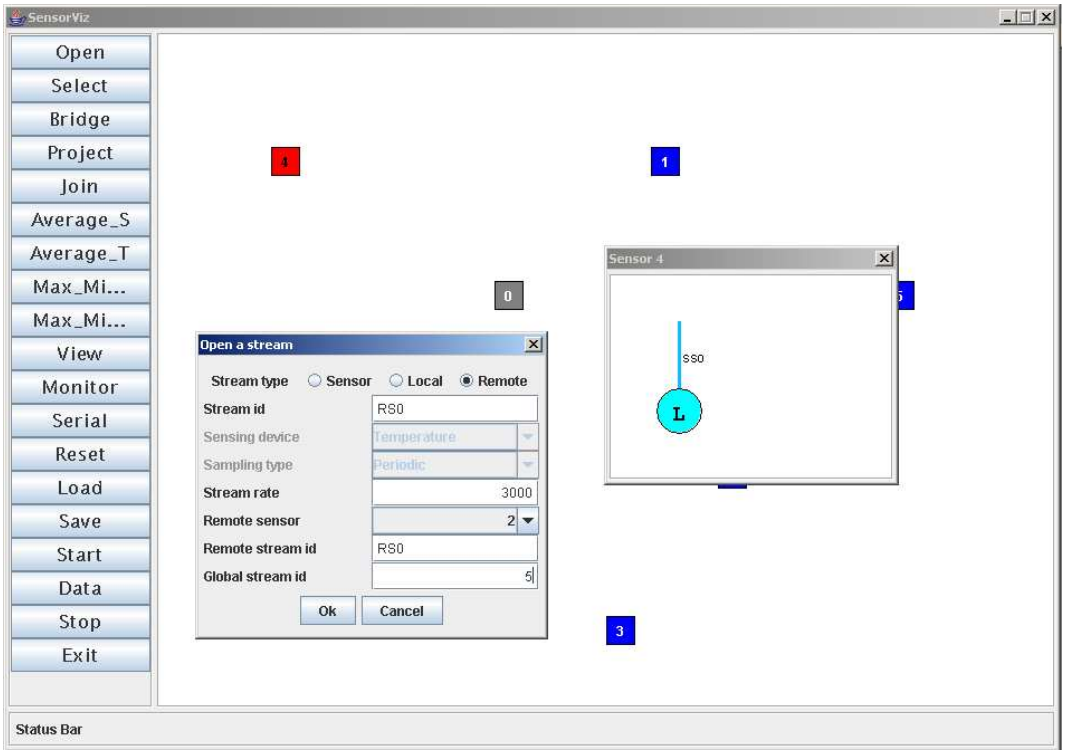


Figura 3.2: Un momento di uso di MUI, l'interfaccia grafica di MaD-WiSe della versione precedente a questo lavoro. Sullo sfondo abbiamo la finestra con a sinistra la barra degli strumenti e a destra il riquadro principale raffigurante la rete di sensori senza fili. In primo piano a sinistra abbiamo la finestra di dialogo usata per creare uno stream ed a destra si ha il riquadro della query del nodo 4.

tra due operatori sullo stesso nodo. Descriviamo ora più in dettaglio questi tre tipi di stream.

3.4.1 Stream sensore

Distinguiamo du tipi di stream sensore:

- **Stream sensore periodico**. Questo tipo si stream è caratterizzato da un sensore e una frequenza di campionamento prefissata a tempo di costruzione del piano di esecuzione della query. Ogni ennupla prodotta avrà come campi:
  - il valore misurato;
  - il tempo in cui è stata effettuata la misurazione, che chiameremo "Timestamp";
  - l'identificatore del nodo in cui è fatta la misurazione (detto "SensorId").

  Il campo "SensorId", poiché ha lo stesso valore per tutte le ennuple dello stream è implicito. Il valore del campo *Timestamp* viene letto dal timer presente nel nodo quando si effettua la misurazione. Lo stesso timer viene usato per far scattare la misurazione al momento giusto, rispettando la frequenza prestabilita.

  Lo stream sensore è associato a un transduttore fisico presente nel nodo, ad esempio:

  Fotometro: (campi: Luce, Timestamp, SensorId);
  Termometro: (campi: Temperatura, Timestamp, SensorId);
  Microfono: (campi: Audio, Timestamp, SensorId);
  Magnetometro: (campi: Magnetismo, Timestamp, SensorId);
  Accelerometro: (campi: Accelerazione X o Y, Timestamp, SensorId).

- **Stream sensore su richiesta**. In questo caso non fissiamo una frequenza di campionamento; la misurazione verrà fatta solo quando richiesto durante l'esecuzione della query.

3.4.2 Stream locale

Nel prossimo paragrafo introdurremo operatori che processano stream e che vengono eseguiti dai nodi della rete. Processare uno stream significa eseguire delle operazioni su ogni ennupla dello stream man mano che arrivano, e restituire una ennupla nello stream in uscita (non necessariamente la stessa ennupla e non necessariamente tutte le volte).

Questi operatori sono collegati fra loro da appositi stream intermedi che si dividono in stream locali e remoti, a seconda se siano sullo stesso nodo o su nodi diversi.

Uno **stream locale** collega due operatori eseguiti dallo stesso nodo. Il primo operatore produce in output lo stream locale, che diventa l'input per il secondo operatore. Uno stream locale è implementato nel query processor eseguito nei nodi come una coda di ennuple. Il primo operatore inserisce l'ennuple prodotte nella coda e l'altro le preleva da essa.

3.4.3 Stream remoto

Uno **stream remoto** è come quello locale, tranne una differenza: i due operatori sono eseguiti su nodi differenti. L'invio di ennuple su uno stream remoto richiede l'uso della rete radio dei due nodi che eseguono gli operatori connessi dallo stream. L'operatore avente come output lo stream remoto, prodotta una ennupla, la invia attraverso la comunicazione radio al nodo di arrivo.Questa trasmissione avviene ad una frequenza prestabilita, l'*output rate* (frequenza di trasmissione) dello stream remoto. Nel nodo di destinazione le ennuple trasmesse vengono ricevute e processate dal relativo operatore che ha lo stream come input.

3.4.4 Nomi dei campi

Sintatticamente utilizziamo una convenzione particolare per i nomi dei campi che compongono l'ennuple che viaggiano negli stream: il nome è preceduto da un prefisso e sono tra loro separati da un punto. Gli stream sensori hanno dei campi con nomi fissati in base al transduttore preso in considerazione, mentre per quelli locali o remoti i nomi vengono decisi dagli operatori che producono in output le ennuple di tali stream. Negli stream sensore il prefisso indica il nodo che ha prodotto il valore dell'attributo che segue tale prefisso per una data ennupla. Ad esempio, $3.Light$ rappresenta il valore della misurazione della luce avvenuta nel nodo 3 della rete di sensori (qualche volta sarà abbreviato con $3.L$). Il campo $Timestamp$ non ha prefisso, perché il tempo di misurazione non è riferito a nessun nodo particolare.

3.5 Algebra su stream

Nei database relazionali tradizionali [2], con il termine algebra si designa in generale un insieme di operatori su relazioni che danno come risultato altre relazioni le quali sono collezioni statiche di dimensione fissata di ennuple, e sono memorizzate in dischi fissi.

Nel sistema MaD-WiSe invece si considerano stream di dati anziché relazioni. La nostra algebra è composta da operatori su stream che danno come risultato altri stream.

Come osservato in [27], una differenza importante tra gli operatori relazionali e quelli su stream è che questi ultimi devono essere "non-bloccanti ($\mathcal{NB}$).

Un'operatore di algebra relazionale si dice bloccante se *'non produce la prima ennupla in uscita prima di aver visto l'intero input.'*

Gli operatori non-bloccanti sono quelli monotonici [27]; un operatore G si dice monotonico quando, dati due collezioni di ennuple A e B:

$$A \sqsubseteq B \Longrightarrow G(A) \sqsubseteq G(B)$$

dove $\sqsubseteq$ è una relazione di ordinamento parziale, $A \sqsubseteq B$ sse A è la parte iniziale di B. In pratica un operatore è monotono se ogni suo output è "definitivo": gli output già prodotti non cambiano se si aggiungono altre ennuple in ingresso. Gli operatori del sistema MaD-WiSe sono tutti di questo tipo (compreso gli aggregati).

3.5.1 Operatori dell'algebra

Si mostrano tutti gli operatori dell'algebra usata, con i quali saranno espressi i piani di esecuzione della query. Sono operatori non-bloccanti su stream.

Useremo solo operatori unari o binari. Un operatore unario prende in input uno stream (sensore, locale o remoto) e restituisce in output un altro stream (locale o remoto); in pratica sia l'input che l'output sono rispettivamente letti e prodotti dall'operatore una ennupla alla volta, con modalità diverse da un operatore all'altro. Invece un'operatore binario ha due stream in entrata e riceverà ennuple da entrambi gli stream in maniera indipendente l'uno dall'altro. Come vedremo un caso particolare è l'operatore binario sync-join che è capace di *sollecitare* ennuple da uno dei propri stream d'ingresso invece che aspettarle passivamente.

Per descrivere formalmente gli operatori useremo le seguenti notazioni:

- con I, S, D rappresentiamo rispettivamente gli stream di input e con O e o' gli stream di output; con $a_1, ..., a_n$ si denotano attributi e con X un insieme di attributi;

- uno stream S concettualmente consiste in una sequenza ordinata non limitata di ennuple t_i che denoteremo con $S =< t_1, ..., t_i >$ con $i \in \mathbb{N}$. ;

- con $t_i[a_j]$ si denota il valore per il campo a_j nell'ennupla t_i;

- un'ennupla t_i è un insieme di coppie *attributo:valore*, che denoteremo con $t_i = \{a_1 : t_i[a_1], ..., a_n : t_i[a_n]\}$;

- con Ts denoteremo l'attributo *Timestamp*, e con $t_i[Ts]$ il relativo valore dell'ennupla i-esima.

- quando due stream S e D hanno lo stesso attributo a_n, per distinguerli si usa la notazione $S[a_n]$ per indicare l'attributo a_n di S e $D[a_n]$ per denotare l'attributo a_n di D.

Proiezione (π)

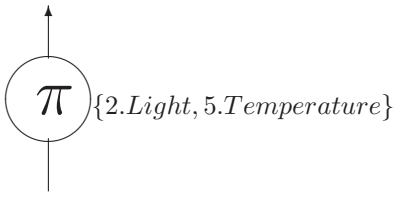
L'operatore di proiezione è unario e per ogni ennupla che riceve dallo stream di ingresso, produce un'ennupla che viene inviata nello stream di uscita. La proiezione sull'insieme di attributi X sarà denotata con π_X. Gli attributi dell'ennupla prodotta sono un sottoinsieme X dei campi di quella in entrata e i relativi valori sono uguali a quello del corrispondente attributo dell'ennupla dello stream di ingresso.

Si noti che il numero di ennuple in uscita è lo stesso di quello in entrata. Formalmente, se $Y = \{a_1, ..., a_n\}$ è l'insieme degli attributi di un'ennupla di un qualunque stream I e $X = \{b_1, ..., b_m\}$ con $X \subseteq Y$, si ha che $\pi_X(I) = O$ dove O è lo stream di output formato da ennuple $u_i = \{b_1 : u_i[b_1], ..., b_m : u_i[b_m]\}$ dove il valore $u_i[b_j]$ per $j \in [1, .., m]$ è quello del corrispondente attributo dell'ennupla di I. Quindi se $I =< t_1, ..., t_i >$ con $i \in \mathbb{N}$ si ha che:

$$
\begin{aligned}
t_i &= \{a_1 : t_i[a_1]\,,\, \ldots\,,\, a_n : t_i[a_n]\} \\
X &= \{b_1, ..., b_m\} \subseteq \{a_1, ..., a_n\} \\
\pi_X(I) &= < u_1, ..., u_i > \quad \text{con } i \in \mathbb{N} \\
&\qquad\qquad \text{e } \; u_i = \{b_1 : t_i[b_1]\,,\, \ldots\,,\, b_m : t_i[b_m]\} \;\text{ con }\; m < n
\end{aligned}
$$

Ad esempio:

2.Light	5.Temperature
5	35
8	32
12	21
...	...



Timestamp	2.Light	3.Audio	5.Temperature
30	5	21	35
33	8	16	32
39	12	12	21
...	...	...	...

Restrizione (σ)

Si tratta di un operatore unario avente come parametro uno stream di ingresso I: quando riceve un'ennupla t_i da I verifica se soddisfa una data condizione c sui valori degli attributi, e nel caso in cui c sia soddisfatta la medesima ennupla t_i è inviata nello stream di uscita, altrimenti t_i è scartata, cioè relativamente ad essa non è mandata nessuna ennupla nello stream di output.

L'operatore di restrizione che usa una condizione c è denotato con σ_c. Pertanto la restrizione $\sigma_c(I)$ restituisce uno stream dello stesso tipo di I e formato dalle ennuple t_i di I che soddisfano la condizione c.

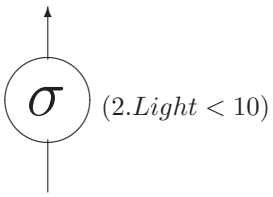
Ogni condizione c è definita con attributi dello stream di ingresso e consiste in un confronto fra un campo dell'ennupla e una costante o fra due campi dell'ennupla. I confronti possibili sono: uguaglianza, minoranza, minoranza stretta, maggioranza, maggioranza stretta e diversità (le condizioni in AND si possono ottenere mettendo le σ in cascata).

Quindi se $I =< t_1, ..., t_i >$ con $i \in \mathbb{N}$, allora

$$
\begin{aligned}
&\sigma_c(I) = O =< u_1, ..., u_j > \\
&\text{dove} \quad \forall u_j \in O \,.\, \exists t_h \in I \quad \text{t.c.} \quad < q(\;\; u_j = t_h \;\; \wedge \;\; c(u_j) = true \;\;)
\end{aligned}
$$

Ad esempio:

Timestamp	2.Light	3.Audio
30	5	21
39	8	16
...	...	...



Timestamp	2.Light	3.Audio
30	5	21
33	12	12
39	8	16
...	...	...

Giunzione sul timestamp ($\bowtie$)

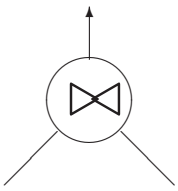
L'operatore di giunzione su stream opera in modo analogo alla giunzione naturale dei database tradizionali su relazioni, ma mentre nella seconda si uguagliano tutti gli attributi con lo stesso nome, nel nostro caso la condizione di giunzione è sempre l'uguaglianza tra gli attributi $Timestamp$.

La giunzione sul $Timestamp$ è un operatore binario, aventi come parametri due stream. L'ennuple che vengono dai due stream di ingresso devono necessariamente avere il campo $Timestamp$. Quando l'operatore riceve due ennuple con lo stesso valore del campo $Timestamp$, una nuova ennupla composta dalla giustapposizione dei campi delle due in arrivo viene prodotta e mandata nello stream di uscita. Ovviamente nell'ennupla in uscita il campo $Timestamp$ compare una volta sola. Quando invece da uno dei due stream in ingresso arriva un'ennupla non corrisposta nel campo $Timestamp$ da alcun'altra ennupla dell'altro stream, questa viene ignorata.

Formalmente, dati due stream S e D con $\{a_1, .., a_n\}$ attributi di S, $\{b_1, .., b_m\}$ attributi di D, a_k e b_h coincidenti con Ts con $k <= n$ e $h <= m$, la giunzione $S \bowtie D$ restituisce uno stream di output O con ennuple ottenute giustapponendo ogni ennupla di S con tutte quelle di D che hanno valori uguali per gli attributi Ts, e con tale attributo presente una sola volta in O. Quindi se $S =< t_1, ..., t_i >$ e $D =< u_1, ..., u_j >$ con i,$j \in \mathbb{N}$ si ha che

$$
\begin{aligned}
t_i &= \{a_1 : t_i[a_1] \, , \, \dots \, , \, a_k : t_i[a_k] \, , \, \dots \, , \, a_n : t_i[a_n]\} \\
u_j &= \{b_1 : u_j[b_1] \, , \, \dots \, , \, b_h : u_j[b_h] \, , \, \dots \, , \, b_m : u_j[b_m]\} \\
S \bowtie D &= \{t_i \circ u_j \mid i, j \in \mathbb{N} \wedge t_i[Ts] = u_j[Ts]\} \\
&\quad \text{e} \\
t_i \circ u_j &= \{a_1 : t_i[a_1] \, , \, \dots \, , \, a_k : t_i[a_k] \, , \, \dots \, , \, a_n : t_i[a_n], b_1 : u_j[b_1] \, , \, \dots \\
&\quad \dots \, , \, b_{h-1} : u_j[b_{h-1}], b_{h+1} : u_j[b_{h+1}] \, , \, \dots \, , \, b_m : u_j[b_m]\}
\end{aligned}
$$

Timestamp	3.Temperature	2.Light
30	28	5
36	5	8
...	...	...



Timestamp	3.Temperature
30	28
36	5
39	2
...	...

Timestamp	2.Light
30	5
33	12
36	8
...	...

Nei database tradizionali la giunzione è spesso implementata in modo bloccante; per esempio se è implementata con l'algoritmo *Nested Loop Join* (NLJ) date le relazioni A e B, per ogni ennupla t_1 della relazione esterna A viene fatta la scansione dell'intera relazione interna B controllando se le relative ennuple soddisfano la condizione di giunzione con t_1. Invece nel sistema MaD-WiSe la giunzione non può essere bloccante perché è eseguita su stream e non su tabelle finite. Gli algoritmi tradizionali per la giunzione richiederebbero, nel caso di stream, buffer di dimensione infinita.

Per implementare la giunzione in maniera non-bloccante si hanno due strategie alternative. Entrambe si basano sul fatto che la condizione di giunzione è sempre fatta sull'attributo Ts che assume valori crescenti.

- **Giunzione continua o progressiva**: Visto che gli stream sono naturalmente ordinati rispetto al Ts, la prima delle due strategie usate è un algoritmo che ricorda in qualche maniera il Merge_join usato nei data base tradizionali. Per questo motivo lo denotiamo *m_join*. L'operatore di giunzione si tiene un buffer di una posizione per ciascun stream di ingresso contenente l'ennupla con il Ts più recente arrivata dal relativo stream. Ogni volta che arriva un'ennupla da uno stream si guarda se l'ennupla nel buffer relativo all'altro stream ha lo stesso valore di Ts. In caso affermativo si effettua la giustapposizione delle due ennuple e si produce l'ennupla in uscita, altrimenti si sostituisce l'ennupla arrivata a quella del buffer del rispettivo stream (l'ultima ennupla arrivata avrà un Ts più recente di quello dell'ennupla già presente nel buffer e che viene sostituita). Chiaramente questo algoritmo funziona se non ci sono ritardi nell'arrivo delle ennuple superiori al passo di campionamento.

- **Giunzione con sincronizzazione**: Uno dei due stream in input (per convenzione il secondo) deve essere uno stream sensore su richiesta. Ogni volta che dall'altro stream si riceve una ennupla t_0 si fa richiesta allo stream sensore di effettuare una misurazione ricevendo l'ennupla s_0; per le ennuple t_0 e s_0, che avranno sempre lo stesso valore di Ts, viene fatta la giunzione come detto sopra e poi vengono spedite spedite. Questa strategia è eseguita dall'operatore $sync\_join$ che denoteremo con $\bowtie_{sync}$.

La seconda strategia è un punto di forza dell'approccio MaDWiSe alle reti di sensori perché consente di non effettuare misurazioni costose quando non è necessario, questo consente un notevole risparmio energetico e permette di prolungare la durata delle batterie dei sensori. Cercare di utilizzare *sync_join* invece che *m_join* ogni volta che è possibile sarà uno degli obiettivi della nostra ottimizzazione (vedi cap. 6). Chiaramente un *sync_join* deve operare nello stesso nodo in cui viene campionato lo stream sensore su richiesta, altrimenti le frequenti richieste di misurazione dovrebbero essere fatte via radio con costo elevato annullando il vantaggio della strategia.

Max e min spaziali (*max*/*min*)

Sia *max* che *min* spaziale sono operatori binari aventi come parametri due stream I_1 e I_2. Tali operatori implementano il calcolo del massimo e minimo spaziali. I due stream in ingresso devono avere i campi su cui si applica l'operatore di aggregazione con gli stessi nomi eccetto che per il prefisso. Indicheremo con max_a (e min_a) l'operatore che calcola il massimo (e il minimo rispettivamente) dell'attributo a, procedendo come segue:

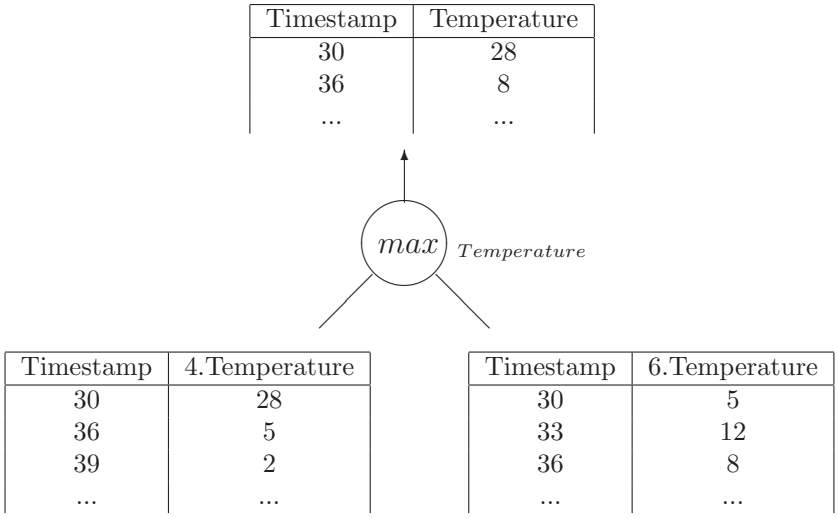
- per ogni ennupla t_i di I_1 avente lo stesso valore di Ts di un'ennupla t'_j di I_2 è calcolato il massimo (o il minimo rispettivamente) tra il valore $t_i[a]$ e $t'_j[a]$;

- nello stream di uscita è inviata un'ennupla contenente Ts e il campo contenente il valore massimo (o minimo rispettivamente) con lo stesso nome ma senza il prefisso del campo dello stream di ingresso su cui è stata valutata l'aggregazione.

Tali operatori sono chiamati spaziali perché aggregano misurazioni provenienti da nodi diversi e quindi situati in posti diversi.

Tramite questi operatori, eseguiti su diversi nodi della rete, si può calcolare il valore dell'aggregato spaziale *in-network* (come in [31]), cioè mentre i dati attraversano la rete dai sensori al nodo sink. Ogni operatore *max* (o *min*) calcola ed inoltra il valore massimo (o minimo) parziale, cioè il massimo visto sino a quel punto nella rete; i valori inferiori non vengono spediti risparmiando così batteria. Alla fine al sink arriverà il massimo (o minimo rispettivamente) valore finale.

Nell'esempio successivo l'ennupla $\{Timestamp : 30, 4.Temperature : 28\}$ ha lo stesso valore di Ts dell'ennupla $\{Timestamp : 30, 6.Temperature : 5\}$ e pertanto l'operatore $max_{Temperature}$ relativamente ad esse invia sullo stream di uscita $\{Timestamp : 30, Temperature : 28\}$.



Media spaziale (avg)

Si tratta di un operatore binario avente come parametri due stream. Il concetto di calcolo nella rete stessa dell'aggregato è simile a quello degli operatori max/min descritti sopra. Anche in questo caso i due stream in ingresso possono avere campi con gli stessi nomi, eccetto che per il prefisso il quale non sarà presente nei campi dello stream di uscita.

La strategia per calcolare la media di un attributo a comporta di dover distinguere nodi che calcolano medie parziali da nodi che calcolano effettivamente la media. Ogni nodo di media parziale dovrà computare la somma parziale dei valori di a visti sinora, e il numero totale di addendi, la cosiddetta molteplicità. Entrambi questi valori, somma e molteplicità, vanno mandati nello stream di uscita in due campi separati e dove l'attributo della molteplicità è indicato col simbolo ♯. Soltanto un apposito nodo finale (denotato con $[f]avg_a$) effettuerà il calcolo della media e manderà in uscita il valore ottenuto senza molteplicità.

L'operatore di media parziale (denotato con $[p]avg_a$) si occupa di sommare i valori ricevuti di ennuple ottenute giustapponendo quelle con Ts uguale e manda nello stream di output un'ennupla con attributi aventi i seguenti valori:

- la somma dei valori ricavati dal rilevamento dei dati dei transduttori;
- il valore del Ts;
- il numero di valori che sono stati sommati.

Gli operatori di media, sia parziali che finali, accettano dagli stream di input ennuple con il campo ♯ (che vengono prodotte da nodi che calcolano medie parziali) oppure anche ennuple senza il campo ♯ (che vengono prodotte da qualunque altro operatore). In quest'ultimo caso il valore della molteplicità viene considerato uguale a uno. Anche questo operatore è chiamato spaziale perché aggrega misurazioni provenienti da nodi diversi.

Formalmente, dati due stream qualunque S e D con $\{a_1, .., a_n\}$ attributi di S, $\{b_1, .., b_m\}$ attributi di D, a_k e b_h coincidenti con Ts con $k <= n$ e $h <= m$,

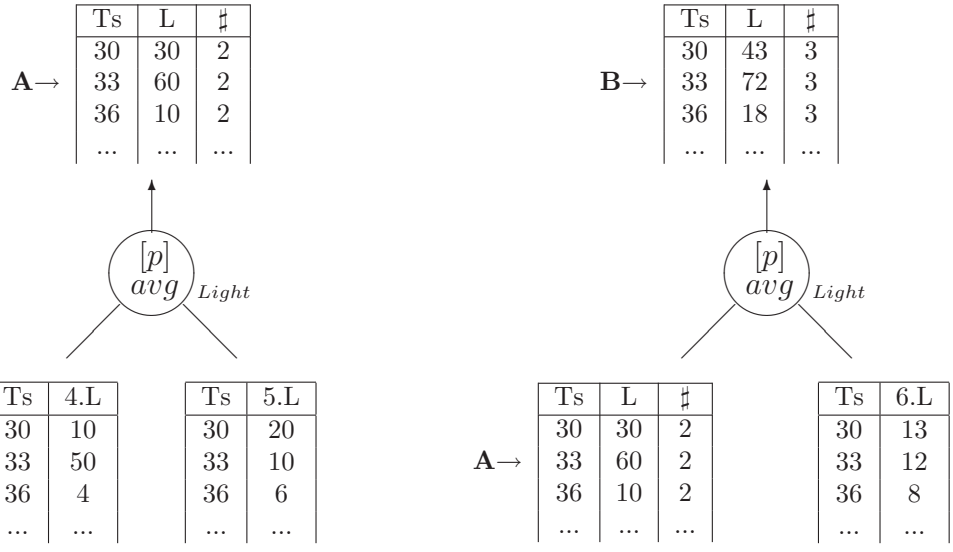
l'operatore $[p]avg_a(S, D)$ restituisce uno stream O formato da ennuple $u_j = \{Ts : u_j[Ts], a : u_j[a], \sharp : u_j[\sharp]\}$ dove:

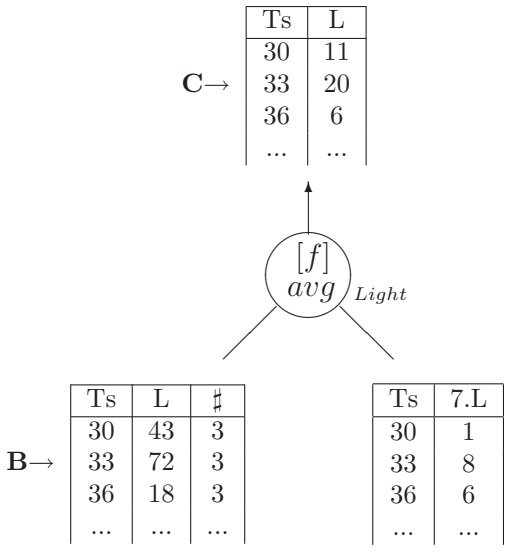
- a è l'attributo di aggregazione comune agli attributi delle ennuple degli stream S e D;
- $u_j[Ts]$ è il valore di Ts relativo alla giustapposizione dell'ennupla di S con quella di D che ha lo stesso valore di Ts;
- $u_j[a]$ è uguale a $S[a] + D[a]$ cioè il valore ottenuto sommando i valori dell'attributo a per tutte le ennuple ottenute giustapponendo quelle di S con D, aventi lo stesso Ts;
- $\sharp$ indica il numero di ennuple che sono state sommate.

Date le ennuple del tipo $u_j = \{Ts : u_j[Ts], a : u_j[a], \sharp : u_j[\sharp]\}$ di O prodotte da un operatore di media parziale e quelle di un qualunque altro stream I con ennuple contenenti l'attributo a, l'operatore $[f]avg_a(O, I)$ restituisce uno stream di output o' con ennuple $z_h = \{Ts : z_h[Ts], a : z_h[a]\}$ dove se v_a è il valore dell'attributo a delle ennuple di I giustapposte con quelle di O, si ha che:

- $z_h[a]$ è il valore dell'espressione $(u_j[a] + v_a)/(u_j[\sharp] + 1)$ se gli attributi delle ennuple di I non contengono $\sharp$, altrimenti $(u_j[a] + v_a)/(u_j[\sharp] + v_\sharp)$ dove $v_\sharp$ è il valore dell'attributo $\sharp$ delle ennuple di I che in tal caso sono il risultato di un'aggregazione parziale;
- $z_h[Ts]$ è il valore di Ts ottenuto giustapponendo ogni ennupla di O con tutte quelle di I che hanno valori uguali per Ts.

Ad esempio, è possibile calcolare la media fra le misurazioni dei quattro fotometri 4.L, 5.L, 6.L e 7.L con due operatori di media parziale e uno di media finale (il risultato è lo stream **C**):





Count Spaziale (*count*)

Si tratta di un operatore binario avente come parametri due stream S e D. Le ennuple dello stream prodotto dall'operatore $count(S, D)$ hanno il campo Ts e il campo $\sharp$, cioè la molteplicità. Il valore di $\sharp$ è calcolato da $count(S, D)$ in modo diverso a seconda che $\sharp$ sia presente o meno tra gli attributi di S e D e quindi che S e D siano stream prodotti a sua volta da un operatore di *count* spaziale.

Per ogni ennupla t_i di S avente lo stesso valore di Ts dell'ennupla z_j di D, $count(S, D)$ produce nello stream di uscita un'ennupla $t = \{Ts : t[Ts], \sharp : t[\sharp]\}$ dove il valore intero $t[\sharp]$ è calcolato come segue:

- se $\sharp$ non è presente tra gli attributi di S e D, allora $t[\sharp]$ è uguale a 2;
- se $\sharp$ è presente solo tra gli attributi di t_i con valore $t_i[\sharp]$, allora $t[\sharp]$ è uguale a $1 + t_i[\sharp]$;
- se $\sharp$ è presente solo tra gli attributi di z_j con valore $z_j[\sharp]$, allora $t[\sharp]$ è uguale a $1 + z_j[\sharp]$;
- se $\sharp$ è presente tra gli attributi di t_i e quelli di z_j, allora $t[\sharp]$ è uguale a $t_i[\sharp]+z_j[\sharp]$.

Aggregati temporali($_{(Timestamp/(epoch\times p))}\gamma_{\{aggr_1(a_1),..,aggr_n(a_n)\}}$)

Gli aggregati temporali calcolano *avg*, *max*, *min* e *count* sui valori degli attributi di ennuple misurate da uno stesso nodo in tempi diversi (a differenza di quelli spaziali che aggregano valori misurati in nodi diversi nello stesso tempo).

Per questo motivo gli aggregati temporali del sistema MaD-WiSe sono simili agli aggregati di un database tradizionale. Tuttavia, in quest'ultimo gli operatori di aggregazione sono di solito implementati in maniera bloccante, nel senso che per poter calcolare una funzione di aggregazione su una tabella è necessario prima

scandire l'intera relazione. Nel sistema MaD-WiSe sarebbe impossibile aspettare "l'ultima ennupla" di uno stream. Quindi per calcolare operatori bloccanti come gli aggregati temporali (ad es. la temperatura massima rivelata in un intervallo di tempo) si sceglie un periodo di tempo allo scadere del quale si produrrà un'ennupla nello stream di uscita aggregando i valori di tutte le ennuple relative a misurazioni fatte durante tale intervallo.

L'operatore di raggruppamento e aggregazione temporale è unario, ha come parametro uno stream ed è denotato con ${}_{(Timestamp/(epoch\times p))}\gamma_{\{aggr_1(a_1),..,aggr_n(a_n)\}}$ dove:

- ogni $aggr_h$ con $h \in [1,..,n]$ è una funzione di aggregazione temporale;

- ogni a_h con $h \in [1,..,n]$ è un attributo;

- *epoch* è la *durata dell'epoca* espressa in numero di campionamenti;

- p è il passo di campionamento, cioè l'intervallo di tempo tra una misurazione e l'altra.

- $(Timestamp/(epoch \times p))$ cioè la *divisione intera* fra l'attributo Ts e la *epoch* moltiplicata per p, è l'espressione di raggruppamento.

Se a_h è l'attributo Ts, nessuna funzione di aggregazione $aggr_h$ si applica ad esso.

L'operatore di aggregazione temporale opera su uno stream di ingresso dove è presente un attributo Ts ed invia nello stream di uscita un'ennupla per ogni intervallo di tempo di ampiezza uguale ad $epoch \times p$ secondi invece che tutte le volte che è ricevuta un'ennupla in ingresso, calcolando le funzioni $aggr_h(a_h)$ con $h \in [1,..,n]$ su tutte le ennuple ricevute in quell'intervallo di tempo. Ogni ennupla dello stream di uscita ha un sottoinsieme $\{a_1...a_n\}$ dei campi in ingresso:

- se a_h è l'attributo Ts, il relativo valore in output è quello riferito all'ultima misurazione nell'intervallo di ampiezza $epoch \times p$;

- ad ogni a_h diverso da Ts è associato il valore della funzione di aggregazione temporale $aggr_h$ (che è una fra *max*, *min*, *avg*, *count*) valutata per quell'attributo su tutte le ennuple ricevute nell'intervallo di ampiezza $epoch \times p$ ed ognuna relativa ad una misurazione con passo p; questo equivale ad aggregare insieme i gruppi di ennuple aventi lo stesso valore di $Timestamp/(epoch \times p)$.

In questo caso non abbiamo bisogno di un attributo molteplicità: la media viene calcolata dividendo la somma dei valori per il numero di letture effettuate nell'intervallo di tempo di ampiezza uguale a $epoch \times p$ secondi. L'operatore di aggregazione temporale è analogo all'operatore di raggruppamento nei database tradizionali, in cui il valore di `GROUP BY` [3] è dato dall'espressione $Timestamp/(epoch \times p)$.

Formalmente, se I è uno stream di input qualunque, si ha che ${}_{(Timestamp/(epoch\times p))}\gamma_{\{aggr_1(a_1),..,aggr_n(a_n)\}}(I) = O$ dove O è lo stream di output formato da ennuple $u_j = \{Ts : u_j[Ts], a_1 : u_j[a_1], ..., a_n : u_j[a_n]\}$ con a_h l'attributo su cui è stata valutata la funzione di aggregazione e $u_j[a_h]$ il relativo valore per $h \in [1,..,n]$, e $u_j[Ts]$ è il valore del Ts relativo all'ultima misurazione dell'epoca. Se

$I =< t_1, ..., t_i >$ con $i \in \mathbb{N}$ e $t_i = \{Ts : t_i[Ts], b_1 : t_i[b_1], \ldots, b_m : t_i[b_m]\}$ con $m \geq n$, allora:

$$_{(Timestamp/(epoch \times p))}\gamma_{\{aggr_1(a_1),...aggr_n(a_n)\}}(I) =< u_1, ..., u_j > \text{ con } j \in \mathbb{N}$$
$$u_j = \{Ts : u_j[Ts], a_1 : aggr_1(a_1), ..., a_n : aggr_n(a_n)\}$$

dove

$$\forall q \in [1, .., n] \ \exists z \in [1, .., m] \text{ t.c. } a_q \text{ ha lo stesso nome dell'attributo } b_z$$

e $aggr_q(a_q)$ è il valore ottenuto valutando la funzione $aggr_q$ su un gruppo di valori di a_q presi da ennuple con stesso valore di $Timestamp/(epoch \times p)$.

Ad esempio:

Timestamp	3.Light	3.Temperature
32	10	35
35	15	12
38	25	23
...	...	...

$(Timestamp/3)$ γ $\{AVG(3.Light), MAX(3.Temp), Timestamp\}$

Timestamp	3.Light	3.Audio	3.Temperature
30	8	21	35
31	10	16	32
32	12	12	21
33	10	4	10
34	15	19	5
35	20	37	12
36	25	40	22
38	25	35	23
...	...	...	...

In [18] vengono presentate e classificate altre strategie alternative di implementazione degli aggregati.

Bridge (⇑)

L'operatore bridge, denotato con ⇑ ed introdotto in [5], è un operatore unario che trasferisce ennuple senza modificarle dal suo stream di input al suo stream di output. Si può dire che l'operatore di bridge sia un'operatore di comodo essendo usato principalmente per far passare ennuple da uno stream sensor a un stream remoto e quindi ad esempio quando un nodo che ha rilevato dei dati deve comunicarli ad un'altro nodo senza elaborarli.

Serial (S)

L'operatore serial, denotato con S prende in input uno stream e non ha nessun output. Le ennuple che arrivano dal suo stream di ingresso vengono mandate dal sink alla base-station, nel nostro caso attraverso la porta seriale, dove verranno visualizzate dall'utente.

Capitolo 4

MW-SQL: un linguaggio di interrogazione per MaD-WiSe

In questo capitolo presentiamo MW-SQL, un linguaggio simile ad SQL per interrogare le reti di sensori e ne mostriamo anche la relativa la grammatica.

4.1 La rete vista come un database

Come abbiamo visto una rete di sensori senza fili ha molti aspetti in comune ma anche alcune differenze importanti rispetto a un database tradizionale o distribuito, come ad esempio:

- data stream illimitati invece di tabelle di dimensione fissa;
- acquisizione di dati con misurazioni in tempo reale invece che lettura di dati preesistenti da un disco;
- elaborazione dei dati in-network invece che in uno o pochi nodi;
- batteria limitata invece che alimentazione di corrente.

Nonostante tutte queste differenze, dal punto di vista dell'utente, una rete di sensori può essere quasi vista come un database. Per questo motivo molti studi [29, 40, 31, 5] hanno evidenziato i benefici di consentire di interfacciarsi alla rete di sensori come ad un database. Ad esempio si può interrogare la rete di sensori con query espresse in un linguaggio simile ad SQL.

Abbiamo definito il linguaggio **MW-SQL**, per il sistema MaD-WiSe descritto. Il linguaggio MW-SQL permette di formulare query che verranno tradotte usando gli operatori dell'algebra di MaD-WiSe. Nell'interfaccia del sistema MaD-WiSe è stata aggiunta una finestra di dialogo per poter introdurre query formulate in MW-SQL (vedi par. 7.2).

Il linguaggio MW-SQL è simile ad SQL, ma come vedremo presenta anche molte importanti differenze. Alcuni costrutti sono cambiati e altri sono stati aggiunti, per gestire le particolarità delle reti di sensori (come ad es. la possibilità di definire il passo di campionamento, la necessità di distinguere fra aggregati spaziali e temporali o l'opportunità di selezionare aree geografiche). Altri costrutti di SQL come NOT

IN, NOT EXISTS, EXCEPT non possono essere consentiti nelle query perché corrisponderebbero ad operatori bloccanti, cioè non restituiscono alcun risultato prima di aver visto l'intero input.

Il linguaggio MW-SQL considera ogni stream sensore corrispondente ad un transduttore in maniera simile a come SQL considera una tabella. In questo caso ci riferiremo a sorgenti, piuttosto che a tabelle. E' anche possibile definire nuovi "sorgenti virtuali" con degli appositi comandi (vedi par. 4.3).

Per illustrare, informalmente, il nostro linguaggio mostreremo degli esempi di query nel prossimo paragrafo. In seguito nel paragrafo 4.4 mostreremo la sintassi del linguaggio formalmente.

MW-SQL non è case sensitive, cioè le sue parole chiave sono riconosciute sia che siano scritte maiuscole che minuscole per rendere la scrittura delle query più agevole. Inoltre, in una query si possono aggiungere dei commenti seguendo la sintassi di Java (sono commenti le stringhe incluse tra `/*` e `*/`, e anche le righe che iniziano con `//`). In questo modo le query possono essere accompagnate da una spiegazione in un linguaggio naturale.

4.2 Il linguaggio MW-SQL: esempi di query possibili

In questo paragrafo mostriamo le capacità del linguaggio MW-SQL con degli esempi significativi.

4.2.1 Query semplici

Se volessimo sapere tutti gli attributi in uscita da uno specifico transduttore (ad es. un fotometro) in uno specifico nodo (ad es. il nodo numero 5) possiamo usare la query:

```
Query1:
SELECT *
FROM 5.Light
```

Come visto nel paragrafo 3.4, dal fotometro esce uno stream che non ha solo il campo `5.Light`, ma anche `5.SensorId` e `Timestamp`. Proprio come in SQL, nella query sopra abbiamo usato $*$ per specificare tutti i campi. Quindi la query1 produrrà la tabella:

Timestamp	**5.Light**	**5.NodeId**
tempo misurazione 1	*misurazione 1*	5
tempo misurazione 2	*misurazione 2*	5
tempo misurazione 3	*misurazione 3*	5
tempo misurazione 4	*misurazione 4*	5
↓	↓	↓
.	.	.
.	.	.
.	.	.

4.2.2 Query con proiezione

Nella clausola `SELECT` della query vista sopra al posto dell'asterisco, si può elencare un sottoinsieme qualunque dei suddetti campi, separati da una virgola.

Ad esempio, se un utente vuole sapere solo il valore `5.Light` allora potrà immettere la query:

Query2:

```
SELECT 5.Light
FROM 5.Light
```

In questo modo misuriamo la luce con il transduttore del nodo etichettato "5" ogni n secondi, dove n è il valore di default del Sampling Rate (ad es. 15 secondi). Ad ogni misurazione il nodo 5 manderà il valore misurato al computer centrale (nodo sink), dove l'utente lo vedrà visualizzato in una tabella (che aumenta la relativa dimensione in tempo reale) composta da una sola colonna etichettata `5.Light`.

5.Light
misurazione 1
misurazione 2
misurazione 3
misurazione 4
↓
.
.
.

4.2.3 Query con scelta del passo di campionamento

Il Sampling Rate, cioè il tempo che intercorre tra una misurazione e la successiva di default vale 3 secondi, ed un valore diverso può essere specificato nella query con la clausola `EVERY`:

Query3:

```
SELECT 5.Light
FROM 5.Light
EVERY 10 SECONDS
```

In questo modo le misurazioni vengono effettuate ogni 10 secondi. Come unità di misura si possono usare millisecondi (`MSECONDS`); se l'unità di misura non viene specificata il numero verrà interpretato come millisecondi. Come sintassi alternativa viene accettato `SAMPLING RATE` invece di `EVERY`.

4.2.4 Query con giunzioni

Dopo la clausola `FROM`, come in SQL tradizionale, si possono specificare molti transduttori diversi, sia dello stesso nodo che di nodi diversi, separati da una virgola.

In questo caso viene fatta la giunzione sull'attributo *Timestamp* di tutti gli stream nella lista (vedi par. 3.5.1). Ad esempio, la query4

Query4:

```
// Campiona il rumore sui nodi 1,2,3
SELECT *
FROM 1.Audio, 2.Audio, 3.Audio
```

produrrà la tabella (aggiornata in tempo reale):

Timestamp	**1.Audio**	**2.Audio**	**3.Audio**
tempo misurazione	*misur 1*	*misur 2*	*misur 3*
tempo misurazione	*misur 1*	*misur 2*	*misur 3*
tempo misurazione	*misur 1*	*misur 2*	*misur 3*
tempo misurazione	*misur 1*	*misur 2*	*misur 3*
↓	↓	↓	↓
.	.	.	.
.	.	.	.
.	.	.	.

Si noti che nel risultato il *Timestamp* è ripetuto una sola volta.

Vediamo un altro esempio che restituisce uno stream con i campi *Timestamp*, `2.MagnetismY`, `2.MagnetismX`:

Query5:

```
// Campiona il campo magnetico (vettoriale) nel
nodo 2
SELECT *
FROM 2.MagnetismY, 2.MagnetismX
```

Come abbiamo visto nel paragrafo 3.5.1, l'operatore di giunzione della nostra algebra è binario: quando la query richiede la giunzione di più di due stream (come nell'es. Query6), di default si associa a sinistra. Attraverso l'uso della funzione esplicita `JOIN`, l'utente può specificare l'ordine con cui devono essere eseguite le giunzioni. Ad esempio:

Query6 alternativa:

```
SELECT *
FROM JOIN(1.Audio, 2.Audio), 3.Audio
```

oppure:

```
SELECT *
FROM 1.Audio, JOIN(2.Audio, 3.Audio)
```

La funzione esplicita `JOIN` può raggruppare un numero qualunque di stream che verranno uniti fra di loro con l'operatore di giunzione binario. Tuttavia se si attiva

l'ottimizzazione topologica, come vedremo nel paragrafo 6.1, il sistema stabilisce l'ordine delle *join* indipendentemente da come richiesto dall'utente.

Anche se negli esempi che abbiamo visto le giunzioni sono eseguite sempre tra stream sensori, in realta' le giunzioni possono essere fatte tra qualunque stream (l'output di una media, o di una inner query, ecc.).

Spesso è utile richiedere la giunzione di tutti i transduttori presenti su un dato nodo. In MW-SQL si possono effettuare query come la seguente:

Query7:

```
SELECT 5.Light, 5.Temperature
FROM 5
```

Supponendo che il nodo 5 contenga transduttori per misurare i campi *Light*, *Temperature*, *Audio*, *AccelerationX*, *AccelerationY*, *MagnetismX* e *MagnetismY*, la clausola `FROM` della query sopra è equivalente a richiedere la giunzione di: `5.Light`, `5.Temperature`, `5.Audio`, `5.AccelerationX`, `5.AccelerationY`, `5.MagnetismX` e `5.MagnetismY`.

Per questo caso, come vedremo nel capitolo sulle ottimizzazioni (in particolare le regole 6.18 e 6.19 del par. 6.2.3), tutti i sorgenti eccetto `5.Light` e `5.Temperature`, saranno riconosciuti inutili e verranno rimossi.

Come altro esempio, consideriamo la query

Query8:

```
SELECT *
FROM 5, 4, 2.Light
```

la quale equivale a richiedere le letture di tutti i transduttori presenti nei nodi 5 e 4, e del fotometro del nodo 2.

4.2.5 Query con restrizione

Come in SQL si possono porre delle condizioni nelle query con la clausola `WHERE`. Ad esempio:

Query9:

```
SELECT 1.Light,2.Light
FROM 1.Light,2.Light
WHERE 1.Light<3
```

In questo caso, si richiedono le misurazioni di luce dei primi due nodi solo quando la luce del nodo 1 risulta essere minore di tre e le ennuple che non soddisfano questa condizione vengono scartate.

Una condizione può consistere in una uguaglianza o in una disuguaglianza (stretta o non) fra un attributo dell'ennupla e una costante, oppure fra due attributi dell'ennupla. Si possono paragonare solo campi con lo stesso nome, eccetto il prefis-

so. Ad esempio: `1.Light<2.Temperature` non è una condizione accettabile, mentre `1.Light<2.Light` lo è.
Più condizioni possono essere combinate in `AND`:

Query10:

```
// Restrizioni a cascata
SELECT 1.Light
FROM 1.Light,1.Audio,2.Audio
WHERE 1.Light<=20
  AND 1.Light>5
  AND 1.Audio>2.Audio
```

Con la suddetta query si vuole sapere la misurazione del fotometro del primo nodo, ma solo quando questa è compresa tra i valori $(5, 20]$ e tale nodo registra più rumore del secondo nodo.

Poiché l'operatore di restrizione della nostra algebra prevede una condizione semplice (vedi par. 3.5.1), allora le condizioni in AND saranno valutate mettendo vari operatori di restrizione in cascata.

4.2.6 Query con aggregati temporali

Il linguaggio MW-SQL permette di richiedere nella clausola `SELECT`, oltre a misurazioni semplici, anche il calcolo di funzioni di aggregazione di n misurazioni successive. Chiameremo queste funzioni di aggregazione "aggregati temporali", perché aggregano misurazioni di uno stesso campo dello stream in un particolare periodo di tempo, detto "epoca". Gli aggregati temporali previsti sono:

- Valore medio (*avg*);
- Valore massimo (*max*);
- Valore minimo (*min*);
- Numero di ennuple (*count*).

La durata dell'epoca può essere specificata con un apposito costrutto. Ad esempio, con la seguente query11

Query11:

```
SELECT AVG(1.Light)
FROM 1.Light
EPOCH 10 SAMPLES
EVERY 20 SECONDS
```

si chiede il valore medio di 10 misurazioni successive (una ogni 20 secondi) della luce del nodo 1. In output si avrà uno stream con solo il campo `1.Light` che presenterà una ennupla ogni 20×10 secondi.

La durata dell'epoca è espressa in numero di passi di campionamento; in questo caso è di 10 intervalli di campionamento.

La parola `SAMPLES`, nella query11, si può omettere per brevità.

In una query si può omettere di specificare sia la durata dell'epoca o il SAMPLING RATE o anche entrambi: in questo caso verranno usati i valori di default, cioè 3 secondi per il SAMPLING RATE e 15 misurazioni per la durata dell'epoca.

Se in una `SELECT` si usano aggregati temporali, allora tutti i campi proiettati devono essere racchiusi in un'operazione di aggregazione. Il campo *Timestamp* fa eccezione e può comparire non aggregato in una `SELECT` che usa aggregati. In pratica gli aggregati temporali in MW-SQL sono corrispondenti alle funzioni di aggregazione standard di SQL di query espresse con la clausola "GROUP BY $Timestamp/(p \times n)$" (usando la divisione intera) con n la durata dell'epoca e p il passo di campionamento. Ad esempio:

Query12:

```
SELECT AVG(3.Light), MAX(3.Temperature),
Timestamp
FROM 3.Light, 3.Temperature
```

Nella suddetta query si richiede la quantità di luce media, la temperatura massima e il tempo di misurazione rilevati nel nodo 3 in ogni epoca. Il *Timestamp* si riferisce all'ultima misurazione dell'epoca.

La query produrrà la tabella (aggiornata in tempo reale):

Timestamp	**3.Light**	**3.Temperature**
tempo 1ma epoca	*luce media epoca1*	*temp max epoca1*
tempo 2nda epoca	*luce media epoca2*	*temp max epoca2*
tempo 3za epoca	*luce media epoca3*	*temp max epoca3*
↓	↓	↓
.	.	.
.	.	.
.	.	.

Nel caso in cui nella query si abbiano delle condizioni, le ennuple che non le soddisfano vengono scartate prima di fare un aggregato temporale. Ad esempio la query13 , richiede la media delle misurazioni dell'audio maggiori di 70:

Query13:

```
SELECT AVG(5.Audio)
FROM 5.Audio
WHERE 5.Audio>70
```

Con la successiva query14 si richiede il numero di volte, nel corso di un'epoca, in cui la temperatura misurata è stata maggiore di 30 gradi:

```
Query14:
SELECT COUNT(5.Temperature)
FROM 5.Temperature
WHERE 5.Temperature>=30
```

Se si vuole applicare le condizioni dopo l'aggregazione (come quando si usa la clausola HAVING in SQL) si può usare una inner query (vedi query27 nel par. 4.2.10).

4.2.7 Query con aggregati spaziali

Nel nostro linguaggio si possono aggregare valori misurati con transduttori dello stesso tipo in nodi diversi. Ad esempio nella query:

```
Query15:
SELECT *
FROM AVG(1.Light,2.Light,3.Light)
```

si richiede la media della luce misurata nei nodi 1, 2 e 3.

In questo caso si tratta di "aggregati spaziali", perché aggregano misurazioni effettuate nello stesso momento ma in posti diversi. Gli aggregati spaziali previsti sono:

- Valore medio (*avg*);
- Valore massimo (*max*);
- Valore minimo (*min*);
- Numero di ennuple (*count*).

Sintatticamente gli aggregati temporali e spaziali si distinguono facilmente perché i primi compaiono nella `SELECT`, mentre i secondi compaiono nella `FROM`. Inoltre quelli temporali operano su un solo attributo, e quelli spaziali su due o più. Per questa ragione si è deciso di utilizzare in MW-SQL, le stesse parole chiavi per i due tipi di aggregati, invece di distinguerli.

Un aggregato spaziale deve prendere stream contenenti campi con nomi uguali del transduttore richiesto eccetto che per prefisso. In uscita produrrà ennuple con campi con lo stesso nome ma senza il prefisso (salvo ridenominazione, vedi par. 4.2.9). Ad esempio la query15 produrrà in uscita:

Timestamp	**Light**
tempo 1	*misurazione media al tempo 1*
tempo 2	*misurazione media al tempo 2*
tempo 3	*misurazione media al tempo 3*
tempo 4	*misurazione media al tempo 4*
↓	↓
.	.
.	.
.	.

Nel successivo esempio con la query16

```
Query16:
// Media di medie
SELECT Audio
FROM AVG(
        AVG(1.Audio,5.Audio,7.Audio),
        AVG(2.Audio,3.Audio)
)
```

si richiede la media tra il valore medio calcolato nei nodi 1, 5, 7 e quello dei nodi 2, 3 relativi alle misurazioni di audio. Ciò può essere utile quando i nodi 1,5,7 sono in una stanza, e i nodi 2,3 in un'altra grande uguale, e si vuole sapere il rumore medio delle due stanze. Chiaramente questo è diverso dal calcolare la media su cinque nodi.

Nel caso in cui nella query si abbiano delle condizioni, dopo aver fatto un aggregato spaziale le ennuple che non le soddisfano vengono scartate.

4.2.8 Query con area e globali

Negli aggregati spaziali e nelle giunzioni è necessario elencare una lista di nodi. Elencare tanti nodi uno per uno non è comodo, specie per reti di sensori molto grandi. Quindi, il nostro linguaggio permette di selezionare un sottoinsieme di nodi in un'area geografica scelta dall'utente.

Per esempio, se supponiamo che la nostra rete sia rappresentata in un'area di 800×600, allora ogni nodo avrà le coordinate (x, y) con $x \in (0, 800)$ e $y \in (0, 600)$. La seguente query

```
Query17:
// si trova la luce media
// dei sensori posizionati
// nel quadrante in alto a sinistra
SELECT *
FROM AVG( AREA(0,0,400,300).Light )
```

è equivalente ad elencare gli stream sensori luce di tutti i nodi aventi coordinate (x, y) con $x \in (0, 400)$ e con $y \in (0, 300)$, e che sono provvisti di un fotometro.

Nella lista dei nodi appartenenti all'area se ne possono aggiungere altri separatamente. Ad esempio, nella successiva query alla lista dei nodi dell'area è aggiunto il nodo 5:

```
Query18:
// si trova la luce massima fra tutti i sensori
// nella metà sinistra, più il sensore 5
SELECT *
FROM MAX(AREA(0,0,400,600).Light, 5.Light)
```

In alternativa si possono fare delle query unendo due o più aree. Nella query19 ad esempio, vengono considerate due aree:

Query19:

```
// si fa la media della luce dei sensori
// posizionati in alto a sx e in basso a dx
// e si selezionano tutti i campi
SELECT *
FROM AVG(AREA(0,0,400,300).Light,
        AREA(400,300,800,600).Light)
```

Il linguaggio MW-SQL mette a disposizione la parola chiave `ALL` che è equivalente a richiedere un'area che copre tutta la rete. Ad esempio per controllare se c'è un incendio, si può scrivere la query:

Query20:

```
// si seleziona la temperatura massima fra
// TUTTI i sensori che
// hanno un termometro
SELECT Temperature
FROM MAX( ALL.Temperature )
```

Come ultimo esempio consideriamo la seguente query

Query21:

```
SELECT *
FROM ALL.Temperature, ALL.Audio
```

che restituisce ennuple con i campi { `Timestamp`, `1.Temperature`, `2.Temperature`, `3.Temperature`, ...,`1.Audio`, `2.Audio`, `3.Audio`, ... }. Si tratta di una query poco pratica (per le dimensioni delle ennuple), ma sintatticamente corretta.

4.2.9 Query con ridenominazione

Qualunque attributo e/o aggregato presente nella clausola `FROM` o dentro qualunque `JOIN`, può essere ridenominato, tramite il costrutto `AS`, con un identificatore scelto dall'utente. Questo identificatore sostituirà il prefisso nel nome di tutti i campi nello stream in questione (eccetto per il *Timestamp*). Per esempio la seguente query

Query22:

```
SELECT *
FROM 5.Light AS Stanza1
```

produrrà la tabella:

Timestamp	**Stanza1.Light**
tempo misurazione 1	*misurazione 1*
tempo misurazione 2	*misurazione 2*
tempo misurazione 3	*misurazione 3*
tempo misurazione 4	*misurazione 4*
↓	↓
.	.
.	.
.	.

Notiamo che il *Timestamp* non viene ridenominato perché non ha mai il prefisso.

La ridenominazione può essere utile soprattutto con gli aggregati spaziali producendo ennuple in cui l'attributo che assume il valore dell'aggregato è privo di prefisso. Per esempio la query

Query23:

```
// Join di due medie
// con la condizione su una delle due
SELECT a.Light, Light
FROM AVG(1.Light, 2.Light) AS a,
    AVG(3.Light, 4.Light)
WHERE a.Light<4
```

non si potrebbe scrivere senza ridenominazione, perché altrimenti non si potrebbe distinguere se l'attributo `Light` usato nella condizione della clausola `WHERE`, si riferisce alle ennuple dovute al calcolo della prima media o a quelle del calcolo della seconda media.

4.2.10 Inner query

In MW-SQL è possibile inserire delle sottoquery (o inner query) al posto di uno stream. Le inner query possono essere opzionalmente messe fra parentesi tonde per disambiguare l'appartenenza di un eventuale `WHERE` alla query o alla inner query (se non viene specificato il comando `WHERE` si riferisce alla `SELECT` immediatamente precedente). Vediamo alcuni esempi:

Query24:

```
SELECT *
FROM ( SELECT *
      FROM 2.Light,3.Light
      WHERE 3.Light<10 ),
     ( SELECT *
      FROM 5.Light,6.Light
      WHERE 6.Light<10 )
```

Query25:

```
SELECT *
FROM (
        SELECT *
        FROM 2.Light,3.Light
        WHERE 2.Light<10)
    ), 1.Light
```

Query27:

```
SELECT 1.Light, Light
FROM AVG(3.Light,4.Light) ,
     SELECT 1.Light,Timestamp
     FROM 1.Light
     WHERE 1.Light<4
```

Nei tre esempi mostrati sopra, quando la condizione della inner query non è verificata, questa non produce ennuple e quindi nemmeno la query esterna.

Nella successiva query, la condizione `1.Light<3` si applica ai valori risultanti dal calcolo della media (come quando si usa HAVING in SQL).

Query28:

```
SELECT *
FROM
  (SELET AVG(1.Light)
   FROM 1.Light)
WHERE 1.Light<3
```

4.3 Sorgenti virtuali

Come abbiamo accennato, è possibile definire delle sorgenti virtuali ed usarle in seguito come le altre sorgenti nelle query. Le sorgenti virtuali sono identificate con un nome scelto dall'utente al momento della creazione, che verrà utilizzato per richiamarle. Dalla stessa finestra di dialogo usata in MUI, in cui si inserisce la query, l'utente potrà anche inserire un comando per definire una nuova sorgente virtuale.

Le sorgenti virtuali definite in una sessione possono essere utilizzate in quelle successive. Per ottenere questo, quando il sistema si accorge che l'utente ha inserito una definizione della sorgente virtuale, il comando inserito viene salvato in coda ad un apposito file di testo "Sources.txt", il quale viene letto all'inizio di ogni esecuzione dell'interfaccia.

4.3.1 Esempi possibili

Come esempio di creazione di due sorgenti virtuali, si può definire "pianouno" e "pianodue" come la lettura di tutti i fotometri dislocati rispettivamente nel primo

e nel secondo piano di un palazzo. Supponendo di sapere che nel "pianouno" si trovano i nodi 5,3,4 e nel "pianodue" quelli 1,2,6, allora possiamo scrivere:

```
Definition1a:
CREATE SOURCE pianouno
AS 5.Light, 3.Light, 4.Light

Definition1b:
CREATE SOURCE pianodue
AS 1.Light, 2.Light, 6.Light
```

Dopo aver dato questa definizione si potrà sapere la media delle luci di ciascun piano, col comando:

```
Query28:
SELECT pianouno.Light, pianodue.Light
FROM AVG(pianouno), AVG(pianodue)
```

Con la suddetta query viene prodotta la seguente tabella:

pianouno.Light	**pianodue.Light**
media misurazioni dei nodi 5,3,4	*media misurazioni dei nodi 1,2,6*
↓	↓
.	.
.	.
.	.

Siccome nel definire "pianouno" (e "pianodue") si è specificato un insieme di stream sensori non aggregati tra di loro, se in una query si usa "pianouno" senza calcolare su esso aggregati, il sistema eseguirà una giunzione tra le ennuple degli stream sensori appartenenti a "pianouno" producendo ennuple coi campi `5.Light`, `3.Light`, `4.Light`, `Timestamp` come nella seguente query:

```
Query29:
SELECT *
FROM pianouno
```

E' anche possibile includere nella definizione di una sorgente virtuale l'operatore con cui si vogliono aggregare le misurazioni eseguite nei nodi. Ad esempio se consideriamo le seguenti definizioni

```
Definition2a:
CREATE SOURCE pianouno
AS AVG(5.Light, 3.Light, 4.Light)

Definition2b:
CREATE SOURCE pianodue
AS AVG(1.Light, 2.Light, 6.Light)
```

allora la query equivalente alla query28 sarebbe stata

```
Query30:
SELECT pianouno.Light, pianodue.Light
FROM pianouno, pianodue
```

Le definizioni di sorgenti virtuali sono flessibili. Ad esempio si può definire "zonaB" come le misurazioni di rumore in una certa area:

```
Definition3:
CREATE SOURCE zonaB
AS AREA(200,0,800,600).Audio
```

Si possono creare anche sorgenti virtuali usando quelle già create. Considerando le definizioni di "pianouno" e "pianodue" date all'inizio, allora possiamo avere

```
Definition4:
CREATE SOURCE piani
AS pianouno, pianodue
```

e si potrà chiedere la luce massima tra i due piani con la query seguente:

```
Query31:
SELECT *
FROM MAX(piani)
```

Una definizione di sorgenti virtuali può riguardare anche una o più inner query, come nel seguente esempio:

```
Definition5:
CREATE SOURCE treFreddo
AS (SELECT *
    FROM 3.Temperature, 3.Audio, 3.Light
    WHERE 3.Temperature<10 )
```

Con la suddetta definizione sarà possibile sapere la luce del nodo 3 solo quando in questo c'è freddo, con la seguente query:

```
Query32:
SELECT treFreddo.Light
FROM treFreddo
```

La parola SOURCE, come scorciatoia sintattica, si può omettere nelle definizioni.

4.4 Grammatica

Il linguaggio proposto utilizza i token (simboli terminali della grammatica) riportati nella tabella 4.1.

```
Parole riservate (keyword):
<CREATE>, <SOURCE>, <SELECT>, <WHERE>, <FROM>,
<AND>, <AVG>, <COUNT>, <JOIN>, <AS>, <SAMPLING>,
<RATE>, <EVERY>, <MAX>, <MIN>, <AREA>, <ALL>,
<EPOCH>, <SAMPLES>, <MSECONDS>, <SECONDS>

Simboli:
< DOT: '.'  >, < VIRGOLA: ',' >, < STAR: '*' >,
< OPEN: '(' >, < CLOSE: ')' >,
< PLUS: '+' >, < MINUS: '-' >,
< LE: '<=' >, < GE: '>=' >, < NE: '!=' >,
< EE: '=' >, < LL: '<' >, < GG: '>' >

Altri token:
< IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >,
< NUMBER: <DIGIT>(<DIGIT>)* >
```

Tabella 4.1: Token usati nel linguaggio MW-SQL.

Le parole riservate sono riconosciute indipendentemente dal case (maiuscolo o minuscolo). Nota che come in Pascal, C o Java gli identificatori possono contenere cifre ma devono cominciare con una lettera. Gli identificatori verranno usati in MW-SQL per identificare i nomi dei campi, le etichette e i nomi dei sorgenti definiti dall'utente.

La grammatica completa di MW-SQL è mostrata nella tabella 4.2.

I simboli non terminali iniziali della grammatica sono `ComandoMW-SQL` e `DefinitionList`:

- quando si esegue un comando dall'interfaccia di MaD-WiSe il parser definito si aspetta un `ComandoMW-SQL`, che può essere un `Definition` oppure una `Query`. Nel primo caso il sorgente virtuale definito viene aggiunto agli altri definiti precedentemente e il comando salvato (vedi par. 4.3), mentre nel secondo caso la query viene interpretata come vedremo nel capitolo successivo;
- quando viene letto il file Sources.txt (vedi par. 4.3), il parser si aspetta una `DefinitionList`.

```
ComandoMW-SQL ::= Definition | Query

DefinitionList ::= (Definition)*

Definition ::=  <CREATE> [<SOURCE>] Var <AS> Source_List

Query ::=  <SELECT> (Select_List|<STAR>)
           FromWhere
           [ <EPOCH> Natural [<SAMPLES>] ]
           [ (<SAMPLING> <RATE>|<EVERY>) Natural
              [<MSECONDS> | <SECONDS>] ]

FromWhere ::= <FROM> Source_List [<WHERE> Condition]

Const ::=[<PLUS> | <MINUS>] <NUMBER>

Natural ::=<NUMBER>

Condition ::=  Select_Item Op ( Select_Item | Const)
               [<AND> Condition]

Op::= <LE> | <EE> | <NE> | <GE> | <GG> | <LL>

Select_List ::=  (
                   Select_Item
                   | (<AVG>|<MAX>|<MIN>|<COUNT>)
                     <OPEN>Select_Item<CLOSE>
                 )
                 [<VIRGOLA> Select_List]

Select_Item ::=  VAR[<DOT> Field]
                 | ( B_Source<DOT> Field )
Source ::=  (<OPEN> Query <CLOSE>)
            | Query
            | (B_Source <DOT> Field)
            | <AVG><OPEN> Aggr_List<CLOSE>
            | <COUNT><OPEN> Aggr_List<CLOSE>
            | <MAX><OPEN> Aggr_List<CLOSE>
            | <MIN><OPEN> Aggr_List<CLOSE>
            | <JOIN><OPEN> Source_List <CLOSE>

Source_List ::=  (Source [<AS>Var]| Area| Var)
                 [ <VIRGOLA> Source_List ]

Aggr_List ::=  (Source | Area | Var)  [ <VIRGOLA> Aggr_List ]

Var ::= <IDENTIFIER>

B_Source ::= <NUMBER>

Field ::= <IDENTIFIER>

Area ::=  <AREA> <OPEN>
            <NUMBER> <VIRGOLA> <NUMBER> <VIRGOLA>
            <NUMBER> <VIRGOLA> <NUMBER>
          <CLOSE> <DOT> Field
          | <ALL> <DOT> Field
```

Tabella 4.2: Grammatica di MW-SQL.

Capitolo 5

Generazione del piano di esecuzione di query MW-SQL

Mostreremo uno schema dell'architettura del sistema in relazione alla generazione del piano di esecuzione delle query che è stato realizzato e come tale architettura abbia degli aspetti simili a quella per l'esecuzione distribuita di query [26]. Si vedrà anche la rappresentazione interna del piano di esecuzione, il parsing della query e il relativo typechecking.

5.1 Metodologie usate

Consideriamo l'architettura nella figura 5.1. Nell'interfaccia di MaD-WiSe è stata inserita una finestra di dialogo attraverso la quale l'utente immette un comando espresso nel linguaggio MW-SQL (vedi cap. 7). Se il comando è una definizione di un sorgente virtuale espresso tramite `CREATE` (vedi par. 4.3), allora la nuova sorgente viene aggiunta in un apposito catalogo, cioè in un file Sources.txt per usi futuri.

Quando il sistema riconosce che il comando è una query definita con `SELECT` (come ad esempio quelle mostrate nel par. 4.2), allora questa viene analizzata e tradotta in una rappresentazione interna corrispondente, che chiameremo piano di esecuzione. Tale rappresentazione costituita da un albero di operatori dell'algebra, è usata nelle fasi successive. La query può contenere i sorgenti virtuali definiti precedentemente con il comando `CREATE`.

Nel paragrafo 5.2 è descritta la rappresentazione interna che useremo, mentre nel paragrafo 5.3 è considerato il processo di creare una rappresentazione durante il parsing della query. Durante questo processo si attua, se richiesto dall'utente, un'ottimizzazione preliminare: l'ottimizzazione topologica (vedi par. 6.1).

Il primo controllo sul piano di esecuzione è quello della sua correttezza di tipo (vedi par. 5.4). Infatti anche se una query rispetta la sintassi della grammatica e viene parsata correttamente non necessariamente genera un piano corretto. Ad esempio si controlla che i transduttori utilizzati siano realmente presenti nei nodi, che gli operatori degli aggregati siano compatibili fra loro, che ogni operatore dell'albero riceva tutti gli attributi richiesti, eccetera. Eventuali errori verranno mostrati all'utente. Durante questa fase si calcoleranno anche alcuni dati che serviranno in seguito, come il tipo dell'output di ogni operatore usato e la sua frequenza di output.

La fase che segue è quella dell'allocazione iniziale degli operatori nei nodi della rete, in cui si decide in quale nodo sarà eseguito ogni operatore (vedi par. 5.5). Questa è un allocazione iniziale che potrà essere modificata in seguito ottimizzandola.

A questo punto il piano di esecuzione è pronto per essere tradotto in un *piano di esecuzione finale*, cioè nella creazione, in vari nodi, di una serie di stream locali, remoti e sensori e di operatori MaD-WiSe che li collegano. Questa fase, descritta nel paragrafo 5.6, nella quale si preparano istruzioni dettagliate per ogni nodo della rete che prenderà parte all'interrogazione, istruzioni che descrivono ad esempio come eseguirla (compreso quali sensori attivare), cosa comunicare ad eventuali altri nodi e quanto spesso. Le istruzioni verranno mandate ai rispettivi nodi.

Opzionalmente, prima della traduzione nel piano di esecuzione finale, il piano può essere ottimizzato per ridurre il consumo di energia, come descritto nel paragrafo 6.2 del prossimo capitolo.

Mediante un comando dell'utente il piano di esecuzione finale viene effettivamente inviato nella wireless sensor network a partire dal nodo sink. La rete di sensori acquisisce i dati e li elabora nella rete stessa come richiesto e invia i risultati al nodo sink che a sua volta li ritrasmette al PC dell'utente, il quale li visualizza attraverso l'interfaccia.

5.2 Rappresentazione interna del piano di esecuzione

La rappresentazione interna usata per il piano di esecuzione di una query è un'albero binario (massimo due figli per nodo) di operatori su stream come quelli visti nel paragrafo 3.5. Le foglie di quest'albero sono stream sensori e i nodi interni sono operatori. Ad ogni nodo si associano delle informazioni addizionali, come la locazione della rete di sensori in cui verrà eseguito e l'insieme di attributi delle ennuple che manderà al nodo padre. Questa rappresentazione serve per essere manipolata facilmente prima di essere tradotta in un piano finale e nello specifico non contiene esplicitamente alcuni aspetti. Innanzitutto, gli stream locali e remoti sono sottointesi: ne sottointendiamo uno fra ogni nodo e ciascun figlio e sarà di tipo locale oppure remoto a seconda dell'allocazione dei due nodi collegati; nella struttura interna si tengono solo puntatori dal padre ai figli e viceversa.

Gli operatori *Bridge* (vedi par. 3.5.1) sono omessi per essere aggiunti dove opportuno al momento della traduzione nel piano finale. Anche il *Serial* (vedi par. 3.5.1) è implicito: ciò significa che la radice dell'albero manda il suo output ad un Serial per farlo visualizzare dall'utente.

Nella rappresentazione interna considerata, non distinguiamo fra le due strategie implementative delle giunzioni, continua o con sincronizzazione (vedi par. 3.5.1). Lo stesso nodo di giunzione verrà implementato in un modo o nell'altro a seconda del contesto: se il figlio di destra è una foglia e sia la giunzione che il figlio destro sono allocati nello stesso nodo, allora la giunzione sarà di tipo *sync_join*, altrimenti sarà di tipo *m_join*. Questo perché, come abbiamo già visto, le giunzioni sincrone sono sempre più convenienti rispetto alle continue e quindi questa strategia va sempre preferita quando è possibile attuarla. Sarà compito dell'ottimizzatore creare configurazioni dell'albero che corrispondono a *sync_join*. Nello stesso modo non distinguiamo esplicitamente nel piano di esecuzione, tra stream sensori su richiesta

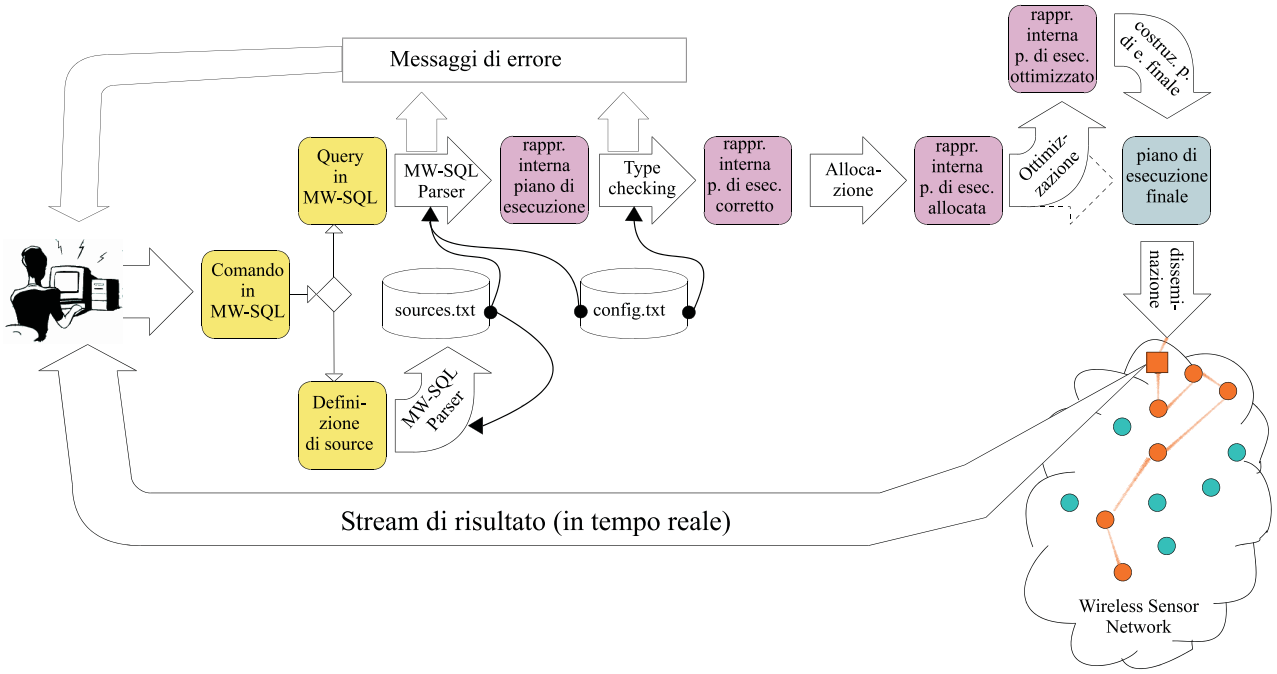


Figura 5.1: Architettura del sistema proposto: l'utente immette il comando nel linguaggio MW-SQL attraverso l'interfaccia e se è una definizione di un sorgente virtuale, allora la nuova source viene aggiunta in un apposito file Sources.txt. Mentre se è una query, viene elaborata e tradotta in una rappresentazione interna del piano di esecuzione. Da questa si controlla il tipo e la correttezza; si trova un'allocazione iniziale degli operatori nei nodi della rete; opzionalmente, si ottimizza il piano. Infine la rappresentazione interna si trasforma nel piano di esecuzione finale che viene disseminato nella WSN a partire dal nodo sink. La rete di sensori acquisisce i dati che li elabora in-network per rimandarli al nodo sink che li ritrasmette all'utente, visualizzandoli attraverso l'interfaccia.

e periodici (vedi par. 3.4.1). Un nodo foglia che sia figlio destro di una giunzione sincrona sarà automaticamente considerato stream sensore su richiesta, gli altri nodi foglia saranno considerati stream sensori periodici.

Il vantaggio delle scelte citate è nella flessibilità della struttura intermedia: basterà modificare la disposizione dei nodi o poche informazioni a loro associate per cambiare in maniera coerente molti aspetti del piano finale.

5.3 Parsing della query e costruzione della rappresentazione interna

Nella fase del parsing, la query in MW-SQL è tradotta nella rappresentazione interna del piano di esecuzione descritta precedentemente. Il parser considerato funziona in maniera standard, come quello dei compilatori di altri linguaggi ad alto livello (vedi [1]). In particolare si tratta di un parser top-down a discesa ricorsiva per la grammatica descritta nel paragrafo 4.4. Il parser è costruito automaticamente con un apposito strumento (vedi par. 7.1), e utilizza un lookahead di cinque token per eliminare il nondeterminismo sulla scelta tra più produzioni.

Eventuali errori pervenuti durante il parsing (errori di lessico o di sintassi) interrompono il processo e vengono mostrati all'utente nell'interfaccia.

Per generare la rappresentazione interna del piano durante il parsing usiamo la tecnica standard:

- ad ogni simbolo non-terminale della grammatica generante il linguaggio MW-SQL si associa un qualche tipo di struttura dati, in modo tale che quando il parser riconosce un simbolo non-terminale produce un'istanza del tipo della struttura dati associata;
- ad ogni regola di derivazione corrisponde una funzione per calcolare l'istanza da produrre utilizzando le istanze prodotte dai non-terminali della parte destra della regola in questione.

Si utilizzano le seguenti associazioni fra simboli non-terminali e tipi di dati:

- **Query:** un albero di operatori (l'albero corrispondente alla query);
- **FromWhere:** un sottoalbero dell'albero degli operatori;
- **Const:** un intero;
- **Natural:** un intero senza segno;
- **Condition:** un sottoalbero composto da una catena aperta di operatori unari σ;
- **Select_List:** un insieme di attributi;
- **Source:** un sottoalbero di operatori (ad esempio il source 1.Light diventa il sottoalbero composto dalla sola foglia 1.Light);
- **Source_List e Aggr_List:** insieme di sottoalberi (che include un sottoalbero di operatori per ogni source della lista);

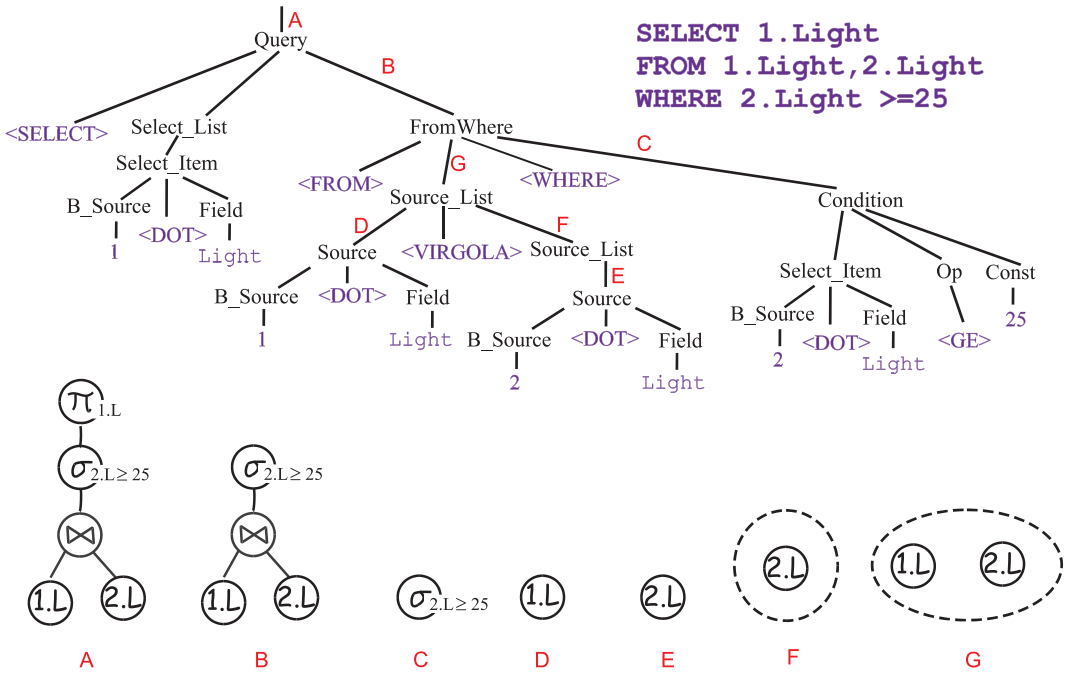


Figura 5.2: L'albero di parsing completo di una semplice query in MW-SQL, mostrata in alto a destra. Per alcuni simboli non-terminali nell'albero abbiamo specificato con delle lettere in rosso la struttura dati prodotta con il riconoscimento di quel simbolo non-terminale. In particolare: A è il piano di esecuzione risultante dal parsing della query, B è un sottoalbero, C'è un sottoalbero incompleto composto da un solo nodo restrizione col figlio non specificato, D ed E sono sottoalberi composti da una sola foglia ciascuno ed F e G sono insiemi di sottoalberi.

- **Var:** un identificatore (una stringa);

- **B_Source:** un identificatore di nodo della rete di sensore (un intero);

- **Field:** un nome di un attributo (una stringa);

- **Area:** una struttura dati apposita per rappresentare un'area rettangolare.

Per brevità, omettiamo di riportare la maggior parte delle funzioni associate ad ogni regola di produzione (vedi fig. 5.2 per un esempio).

Una caratteristica importante del nostro sistema è il trattamento del caso dei simboli non-terminali Source_List e Aggr_List. Come si vede dalla grammatica queste liste sono composte da source, aree e variabili. Il corrispondente insieme di sottoalberi viene composto unendo tutti i contributi di ogni elemento della lista, in particolare:

- per i source singoli si aggiunge il sottoalbero corrispondente;
- per le aree si aggiungono tutti i sottoalberi ciascuno composto da un nodo foglia corrispondente ad ogni nodo della rete di sensore dentro l'area;
- per le variabili si aggiungono tutti i sottoalberi dell'insieme associato al corrispondente sorgente virtuale.

Quando bisogna trasformare un insieme di sottoalberi in un singolo sottoalbero, tutti i sottoalberi dell'insieme verranno uniti tramite una serie di operatori binari: nella figura 5.3 a tale proposito è mostrato un esempio con la giunzione. In tal caso quando il parser riconosce il simbolo non-terminale `Source_List` costruisce l'insieme di sei sottoalberi mostrati nella figura in basso a sinistra ed ottenuti come unione dei contributi dei quattro elementi della lista:

- A è un source singolo che produce un albero composto da una sola foglia;
- B è un'area che racchiude due nodi della rete che hanno il fotometro richiesto;
- C è un sorgente virtuale, precedentemente definito come una coppia di source col comando `CREATE` mostrato in cima nella figura;
- D è una inner query che produce un albero di due nodi.

Quando il `Source` è riconosciuto con la regola `Source::= <JOIN><OPEN> Source _ List <CLOSE>`, ad esso è associato l'albero in basso a destra ottenuto incollando insieme i sei sottoalberi con alcuni operatori binari di giunzione. Nell'esempio questo passaggio è effettuato applicando l'ottimizzazione topologica per minimizzare le distanze tra i nodi della rete che dovranno comunicare (vedi par. 6.1). Da notare che ad esempio, i due sottoalberi $4.A$ e $\sigma_c(4.T)$ pur provenendo da parti diverse della query, sono situati sotto la stessa giunzione e che le due foglie del sorgente virtuale `pippo` non sono vicini nell'albero finale. Invece, nel caso della regola di produzione `Source::=<AVG><OPEN> Aggr_List<CLOSE>`, il corrispondente insieme di sottoalberi è trasformato in un singolo sottoalbero tramite una serie di operatori binari relativi ad aggregati di media spaziale. Un'altra trasformazione di un insieme di sottoalberi in un unico albero, si ha per il non-terminale `FromWhere` per l'esempio mostrato nella figura 5.2.

Ci sono molte alternative per trasformare un insieme di sottoalberi in un singolo sottoalbero, che determinano sequenze diverse di comunicazione tra nodi. Tra queste ce ne saranno alcune più convenienti: quelle che fanno comunicare fra loro nodi vicini. In questa fase si può attuare dunque un'ottimizzazione preliminare, che chiamiamo ottimizzazione topologica, per trovare una buona soluzione (descritta nel par. 6.1 del capitolo sull'ottimizzazione).

Alcuni esempi di risultato della costruzione della rappresentazione interna a partire da una query saranno mostrati nel paragrafo 7.4.

5.4 Typechecking e determinazione degli output

Anche se il parsing di una query non dà nessun errore e quindi la query è corretta sintatticamente, non è detto che essa sia corretta semanticamente.

In questa fase eventuali altri errori possono essere scoperti analizzando il piano di esecuzione risultante dal parsing. Per ogni operatore abbiamo dei **vincoli di correttezza** che devono essere rispettati, ad esempio ogni operatore di proiezione può selezionare solo un sottoinsieme degli attributi che riceve dallo stream di input. In caso contrario l'analisi della query viene interrotta e gli errori verranno mostrati dettagliatamente all'utente. è evidente che per sapere se un operatore rispetta i

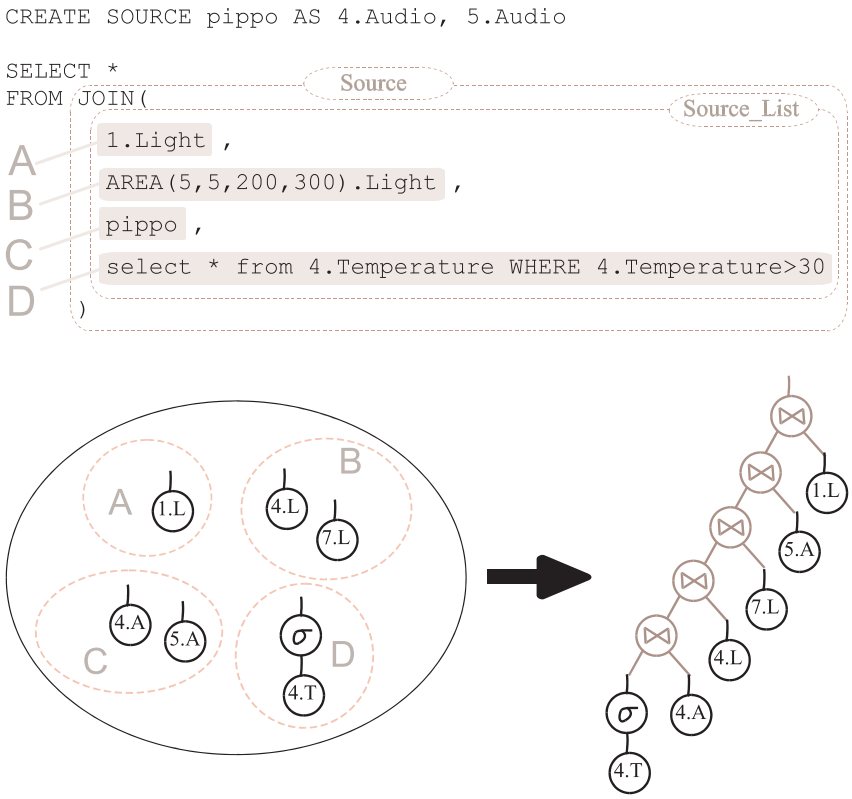


Figura 5.3: Uno dei passaggi della traduzione di una query in una rappresentazione interna. Quando il parser riconosce il simbolo non-terminale `Source_List` costruisce l'insieme di sei sottoalberi mostrati nella figura in basso a sinistra ed ottenuti come unione dei contributi dei quattro elementi della lista: A è un source singolo che produce un albero composto da una sola foglia; B è un'area che racchiude due nodi della rete che hanno il fotometro richiesto; C è un sorgente virtuale, precedentemente definito come una coppia di source col comando `CREATE` mostrato in cima nella figura; D è una inner query che produce un albero di due nodi. Quando il `Source` è riconosciuto con la regola `Source::= <JOIN><OPEN> Source_List <CLOSE>`, ad esso è associato l'albero in basso a destra ottenuto incollando insieme i sei sottoalberi con alcuni operatori binari di giunzione.

vincoli o no è necessario conoscere l'insieme di attributi dei suoi stream di input. Inoltre in questa fase si calcola l'**output** di ogni nodo del piano di esecuzione (a partire dalle foglie). Questa informazione è utilizzata anche nella successiva fase di ottimizzazione.

Un'altro dato utile nelle fasi successive sarà l'**output rate** di ogni nodo del piano, cioè la frequenza massima con cui quel nodo manderà ennuple al nodo padre.

Ora per ogni operatore specifichiamo quali vincoli di correttezza si devono controllare, come si calcola l'insieme degli attributi di output (a meno di ridenominazione), come si calcola l'output rate. Tenendo presente che nel caso in cui al nodo in questione sia stata associata un'etichetta di ridenominazione, questa sostituisce il prefisso di tutti gli attributi nell'insieme dell'output calcolato per quel nodo, si ha:

- ***Nodi foglia**:*
  - **vincoli di correttezza:** il nodo della rete di sensori con l'identificatore richiesto deve esistere e deve avere il transduttore specificato. Queste informazioni sono ottenute dal sistema leggendole da un file di configurazione che descrive lo stato fisico della rete di sensori;
  - **output:** gli attributi in uscita sono quelli prodotti dal transduttore richiesto, come descritto nel paragrafo 3.4.1. Anche queste informazioni sono ottenute dal sistema leggendole da il file di configurazione;
  - **output rate:** è usato solo per gli stream sensori periodici e per definizione l'output rate è quello specificato dall'utente nella query con la clausola `EVERY` (vedi par. 4.2.3), o quello di default se non specificato.
- ***Nodi Proiezione (π_X):***
  - **vincoli di correttezza:** l'insieme di attributi proiettati X deve essere un sottoinsieme (anche non stretto) dell'output del figlio;
  - **output:** è l'insieme X;
  - **output rate:** è l'output rate del figlio; poiché verrà prodotta un'ennupla per ogni ennupla processata.
- ***Nodi Restrizione (σ_c):***
  - **vincoli di correttezza:** la condizione c deve utilizzare solo attributi presenti nell'output del figlio. Inoltre, se la condizione confronta due attributi fra loro, allora questi devono essere dello stesso tipo, cioè devono avere lo stesso nome di campo perché transduttori diversi producono dati espressi in differenti unità di misura. Ad esempio, $1.Temperature \leq 5.Temperature$ è una condizione corretta, ma per $1.Temperature \leq 5.Audio$ si ha un errore di tipo;
  - **output:** è lo stesso insieme di attributi di output del figlio;
  - **output rate:** è l'output rate del figlio. Infatti dobbiamo prevedere l'output rate massimo possibile, che si verifica quando tutte le ennuple processate verificano la condizione (caso che non possiamo escludere).
- ***Nodi Aggregato Temporale ($_{(Timestamp/(epoch\times p))}\gamma_{\{aggr_1(a_1),..,aggr_n(a_n)\}}$):***

- **vincoli di correttezza:** l'insieme $\{a_1, ..., a_n\}$ deve essere un sottoinsieme (anche non stretto) dell'output del figlio. Ogni attributo, eccetto al più il $Timestamp$, deve comparire dentro un aggregato temporale; infatti non è consentito mettere aggregati e non aggregati nella clausola `SELECT`;
- **output:** consiste nell'insieme $\{a_1, ..., a_n\}$;
- **output rate:** dato il funzionamento di questo operatore (vedi par. 3.5.1), l'output rate è quello specificato dall'utente nella query con la clausola `EVERY` (vedi par. 4.2.3) moltiplicato per la durata dell'epoca specificata con la clausola `EPOCH` (vedi par. 4.2.6).

- ***Nodi Giunzione*** ($\bowtie$)***:***
  - **vincoli di correttezza:** siano $O(A)$ e $O(B)$ rispettivamente l'insieme di output del figlio di destra A e di sinistra B. Allora $Timestamp$ deve appartenere ad entrambi $O(A)$ e $O(B)$, perché la condizione di giunzione è una uguaglianza tra il $Timestamp$ delle ennuple di A con il $Timestamp$ delle ennuple di B. Inoltre $O(A) \cap O(B)$ non deve contenere nient'altro che il $Timestamp$, altrimenti nel risultato ci sarebbero due campi con lo stesso nome;
  - **output:** l'output è l'unione dei due insiemi di attributi di output dei figli cioè $O(A) \cup O(B)$;
  - **output rate:** è il massimo degli output rate dei figli, perché il nodo giunzione produce un'ennupla solo quando entrambi i figli forniscono un'ennupla al padre.

- ***Nodi Max/Min Spaziale*** (Max/Min)***:***
  - **vincoli di correttezza:** si hanno i vincoli specificati nel caso del nodo di giunzione. Inoltre gli attributi dell'insieme di output del figlio di sinistra e di quello di destra devono avere lo stesso nome di campo indipendentemente dal prefisso e contenere l'attributo per il quale si vuole calcolare il massimo o il minimo. Infatti ad esempio eseguire un calcolo di massimo spaziale fra un valore di Audio (misurato in decibel) e uno di Temperatura (misurato in gradi) è un errore di tipo;
  - **output:** l'output contiene il $Timestamp$ e gli attributi, senza alcun prefisso, su cui si effettua il massimo o il minimo;
  - **output rate:** si calcola come nel caso dei nodi giunzione.

- ***Nodi Avg Spaziale*** (avg)***:***
  - **vincoli di correttezza:** si hanno i vincoli specificati nel caso come nel caso dei nodi Max/Min Spaziale;
  - **output:** l'output contiene il $Timestamp$ e l'attributo per il quale è stata calcolata la media con l'aggiunta dell'attributo molteplicità $\sharp$ (vedi par. 3.5.1) per i nodi di media parziale;
  - **output rate:** si calcola come nel caso dei nodi giunzione.

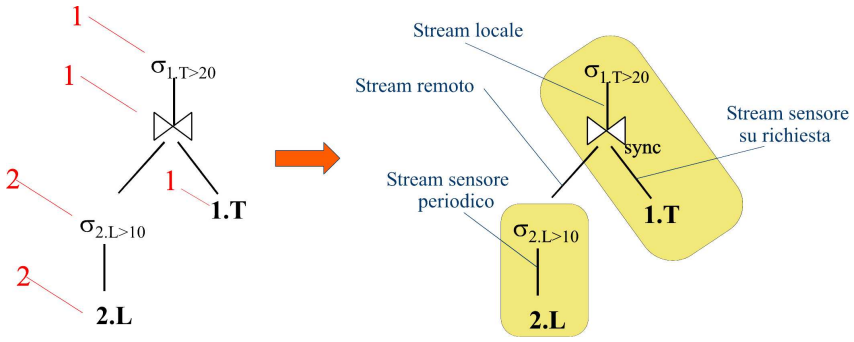


Figura 5.4: Esempio di una allocazione del piano di esecuzione (albero a sinistra) che determina il tipo dei vari stream e la strategia di giunzione sincrona.

- ***Nodi Count Spaziale*** *(COUNT):*

  - **vincoli di correttezza:** si hanno i vincoli specificati nel caso del nodo di giunzione;
  - **output:** è l'insieme costituito dagli attributi *Timestamp* e molteplicità;
  - **output rate:** si calcola come nel caso dei nodi giunzione.

5.5 Allocazione iniziale

In questa fase dobbiamo assegnare ogni nodo dell'albero rappresentante il piano ad un nodo della rete di sensori. Questo equivale a decidere in quale locazione della rete avverranno le elaborazioni in-network dei dati. Da questa scelta dipenderà anche il tipo dei vari stream impliciti e la strategia di giunzione che verrà applicata (vedi fig. 5.4).

L'ottimizzatore nella fase successiva potrà cambiare, l'assegnamento iniziale dei nodi dell'albero ai nodi della rete, ma abbiamo comunque bisogno di una allocazione iniziale. Infatti come vedremo l'ottimizzazione (che comunque è facoltativa) è realizzata con un'euristica greedy che ha bisogno di partire da una soluzione valida, meglio ancora se essa è già una buona soluzione dal punto di vista del consumo di energia.

Un vincolo che deve essere rispettato in ogni allocazione valida è che le foglie devono stare nel nodo della rete di appartenenza del transduttore corrispondente.

L'allocazione iniziale è effettuata con l'algoritmo ricorsivo bottom-up mostrato nella tabella 5.1.

Ad esempio nel caso di allocazione della figura 5.4, poiché l'operatore di restrizione è unario, l'algoritmo alloca la restrizione $\sigma_{2.L>10}$ nel nodo due della rete relativo al transduttore *Light*.

Con l'algoritmo mostrato nella tabella 5.1 si minimizza il numero di stream remoti e si dà la possibilità alle giunzioni di applicare la strategia di sincronizzazione allocando l'operatore di giunzione nel nodo della rete in cui è stato allocato il relativo figlio destro.

```
Procedure Locate(n:nodo)
begin
  if (n è foglia)
    then n.locazione ← locaz del transdut
corrisp;

  if (n è operat unario) then begin
    Locate(n.figlio);
    n.locazione ← n.figlio.locazione;
  end

  if (n è operat binario) then begin
    Locate(n.figliosx);
    Locate(n.figliodx);
    n.locazione ← n.figliodx.locazione;
  end
end
```

Tabella 5.1: Algoritmo di allocazione iniziale.

5.6 Costruzione del piano di esecuzione finale

Una volta effettuata l'allocazione e il typechecking (ed eventualmente l'ottimizzazione), il piano di esecuzione può essere tradotto in un piano finale composto da una serie di stream e operatori di MaD-WiSe.

La creazione del piano di esecuzione finale è fatta utilizzando la parte centrale del sistema di MaD-WiSe (la stessa che precedentemente era utilizzata direttamente dall'utente attraverso la vecchia interfaccia descritta nel par. 3.3 e in [5]). Secondo questo sistema, gli stream (locali, sensori e remoti) devono essere creati prima degli operatori che li collegano.

La strategia di costruzione che sarà usata consiste nel visitare ricorsivamente l'albero del piano di esecuzione creando prima tutti gli stream (dalla radice alle foglie "top-down") e poi creando tutti gli operatori (dalle foglie alla radice "bottom-up").

Descriveremo in pseudocodice la funzione che fa quanto detto sopra, utilizzando le seguenti sottofunzioni per costruire stream:

- `Function costruisci_stream_sensore(n:nodofoglia):stream`
  Crea uno stream sensore che rispecchia i dati di n e quindi la locazione sarà il nodo della rete n.locazione, il sampling rate sarà n.outputrate, il tipo sarà quello di n e la strategia di campionamento -su richiesta o periodico- dipenderà dalle condizioni descritte nel paragrafo 5.2. Restituisce lo stream sensore creato;

- `Function costruisci_stream_remoto(a,b:nodeId; freq:intero):stream`
  Costruisce e restituisce uno stream remoto che collega i nodi della rete di sensori di indice a e b (da a verso b) e che ha una frequenza di trasmissione `freq`;

- `Function costruisci_stream_locale(a:nodeId):stream`

Costruisce e restituisce uno stream locale all'interno del nodo della rete di sensori di indice a.

Useremo anche le seguenti sottoprocedure per costruire operatori:

- `Procedure costruisci_op_serial(a:nodeId; s_input:stream)`
  Costruisce un nuovo operatore di tipo serial (vedi par. 3.5.1) nel nodo della rete di sensori di indice a, che ha s_input come stream di ingresso;

- `Procedure costruisci_op_bridge(a:nodeId; s_input,s_output:stream)`
  Costruisce un nuovo operatore di tipo bridge (vedi par. 3.5.1) nel nodo della rete di sensori di indice a, che ha s_input come stream di ingresso e s_output come stream di uscita;

- `Procedure costruisci_op_unario(n:nodo; s_input,s_output:stream)`
  Il parametro n è un nodo unario del piano, cioè una proiezione, selezione o aggregato temporale. La procedura crea l'operatore unario corrispondente di MaD-WiSe che rispecchia i dati di n, utilizzando s_input come stream di ingresso e s_output come stream di uscita i quali possono essere stream di qualunque tipo. L'operatore sarà creato nel nodo della rete n.locazione;

- `Procedure costruisci_op_binario(n:nodo;`
  `s_input_sx,s_input_dx, output:stream)`
  Come sopra, ma la procedura crea operatori binari. Il parametro n è un nodo binario del piano, cioè una giunzione o uno qualunque degli aggregati spaziali. Gli stream s_input_sx e s_input_dx saranno usati come stream di ingresso dell'operatore creato.

Inoltre useremo la funzione `costruisci_stream_output`, di cui mostriamo lo pseduocodice nella tabella 5.2. Questa funzione costruisce lo stream uscente di un nodo preoccupandosi di crearlo di tipo locale, remoto o sensore a seconda delle esigenze e si occupa anche di creare un bridge se necessario.

La funzione per tradurre un sottoalbero della rappresentazione interna nella parte corrispondente del piano di esecuzione finale scritta in modo ricorsivo è mostrata nella tabella 5.3.
Al piano finale manca il nodo Serial (vedi par. 3.5.1).

Per tradurre la rappresentazione interna nel piano di esecuzione finale utilizzeremo una singola chiamata alla funzione `ToMadwiseRic` tramite la procedura `ToMadwise` illustrata nella tabella 5.4.

Esempi del risultato dell'esecuzione di questo algoritmo saranno mostrati nel paragrafo 7.4 e in particolare nelle figure 7.5, 7.3 e 7.4.

Una volta che il piano di esecuzione è stato tradotto in operatori e stream finali, l'utente può visualizzarli graficamente attraverso l'interfaccia di MaD-WiSe. Infatti, come abbiamo visto nel capitolo 3, ad ogni stream e operatore creato corrisponde un oggetto grafico nel pannello associato al rispettivo nodo. L'utente si può così fare un'idea del piano di esecuzione finale risultante dalla query che ha immesso prima di eseguirlo.

```
// dato un nodo, costruisce e restituisce
// lo stream che va verso il padre

Function costruisci_stream_output(n:nodo):stream
var ss,rs,ls:  stream; // uno stream sensore, remoto e
locale
var start,dest:  interi; // nodi di partenza e destinazione
begin

  start ← n.locazione;
  if (n ha padre) then dest ← n.padre.locazione;
                else dest ← Id del nodo sink

  if (n è foglia) then begin
    ss ← costruisci_stream_sensore(n);
    if (start=dest) then return ss;
    else begin
     rs ← costruisci_stream_remoto(start,dest,
n.outputrate);
     costruisci_op_bridge(start, ss, rs);
     return rs;
    end;
  end;

  else begin // n non è foglia
    if (start=dest) then begin
     ls ← costruisci_stream_locale(start);
     return ls;
    end else begin
     rs ← costruisci_stream_remoto(start,dest,
n.outputrate);
     return rs;
    end;
  end;
end
```

Tabella 5.2: Pseudocodice della funzione `costruisci_stream_output`

```
// costruisce il piano di esec.  finale
// per il sottoalbero di radice n e
// restituisce lo stream che da n va verso il padre

Function ToMadwiseRic(n:nodo):stream
var s_input, s_output, s_input_sx, s_input_dx :  stream;
begin

  if (n è foglia)
    then return costruisci_stream_output(n);

  if (n è operat unario) then begin
    s_output ← costruisci_stream_output(n);
    s_input ← ToMadwiseRic(n.figlio);
    costruisci_op_unario(n, s_input, s_output);
    return s_output;
  end

  if (n è operat binario) then begin
    s_output ← costruisci_stream_output(n);
    s_input_sx ← ToMadwiseRic(n.figlio_sx);
    s_input_dx ← ToMadwiseRic(n.figlio_dx);
    costruisci_op_binario(n, s_input_sx, s_input_dx,
s_output);
    return s_output;
  end

end
```

Tabella 5.3: Algoritmo ricorsivo di creazione del piano di esecuzione finale di un sottoalbero.

```
// traduce in piano di esec finale
// il piano che consiste nell'albero di radice n

Procedure ToMadwise(n:nodo)
var s :  stream;
begin

  s ← ToMadwiseRic(n);
  costruisci_op_serial(Id del nodo sink, s);
end
```

Tabella 5.4: Procedura per la creazione del piano di esecuzione finale data una rappresentazione interna del piano.

Capitolo 6

Ottimizzazione

Precedentemente abbiamo visto come si costruisce un piano di esecuzione della query a partire da essa espressa nel linguaggio MW-SQL ottenuto adattando SQL. Abbiamo anche visto come questo piano venga trasformato in una serie di operatori localizzati nei vari nodi della rete, disseminato ed eseguito. Tra la fase di creazione del piano e quella della relativa disseminazione è possibile eseguire un'ottimizzazione del piano che sarà trattata in questo capitolo.

Daprima ci occuperemo dell'*ottimizzazione topologica*, cioè ottimizzare i percorsi fisici che i dati fanno dal punto in cui vengono rivelati al sink (attraverso altri nodi mediante l'approccio multi-hop). Questa ottimizzazione viene eseguita durante la creazione del piano stesso. Successivamente considereremo l'*ottimizzazione del piano di esecuzione* dove affronteremo il problema di ottenere un piano di esecuzione equivalente con costi minori.

6.1 Ottimizzazione Topologica

6.1.1 Obiettivi

L'obiettivo dell'ottimizzazione topologica è far si che le comunicazioni avvengano tra nodi geograficamente più vicini possibili in modo da minimizzare le interferenze, i fallimenti di trasmissione e l'energia necessaria a trasmettere.

Infatti più due nodi che comunicano fra loro sono lontani, più la comunicazione è difficile e dispendiosa di tempo ed energia, perché spesso può accadere che la trasmissione di dati oppure il messaggio di avvenuta ricezione (acknowledgement) si perdano e la trasmissione sarà quindi ripetuta automaticamente. Inoltre se i nodi sono al di là del raggio di comunicazione sarà necessario l'intervento di altri nodi secondo la modalità di comunicazione multi-hop.

6.1.2 Come e quando si esegue

L'ottimizzazione topologica viene fatta durante la fase di costruzione della rappresentazione interna del piano di esecuzione, quando si legano insieme tante sottoespressioni attraverso alberi binari (con operatori *join*, *max*, *min* e *avg*) come spiegato nel paragrafo 5.3. Ad esempio, supponendo di volere calcolare $avg(1.L, 2.L, 4.L, 5.L)$ si dovrà mettere insieme quattro sottoespressioni composte da una foglia ciascuna

$(1.L)$, $(2.L)$, $(4.L)$ e $(5.L)$ attraverso operatori avg binari, avendo così:
$avg_f\,(\,avg_p\,(\,avg_p\,(\,1.L\,,\,2.L\,)\,,\,4.L\,)\,,\,5.L\,)$. Prima di fare questa operazione l'ottimizzazione topologica riordina queste sottoespressioni per ottenere un percorso minimo.

6.1.3 Minimizzazione della lunghezza dei path

Vediamo l'algoritmo PERCORSO MINIMO per determinare il cammino di lunghezza minima nella rete:

sia $M = \{m_1, \ldots, m_n\}$ un insieme di n mote che devono far parte della query e m_1 sia il sink. Supponiamo che un nodo riceva messaggi da al più un'altro nodo. Allora l'albero di routing diventa un percorso che inizia dal nodo m_1 e finisce in un nodo qualunque m_k passando per tutti gli altri: si deve trovare questa sequenza di nodi. Dobbiamo determinare una permutazione $P =< p_1, \ldots, p_n >$ t.c. il primo elemento sia $p_1 = 1$ e sia minima la seguente somma

$$\sum_{i=1}^{n-1} dist(m_{p_i}, m_{p_{i+1}}) \tag{6.1}$$

dove $dist$ è la distanza euclidea fra due mote, cioè

$$dist(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

. In questo modo si fanno comunicare i nodi tra loro più vicini minimizzando i costi di trasmissione, le interferenze e diminuendo i casi in cui due nodi, fuori dal loro range di comunicazione, debbano comunicare.

Siccome il numero di permutazioni possibili è $n!$, allora sarebbe troppo costoso provarle tutte per minimizzare la (6.1). Si usa, invece, l'euristica mostrata nella tabella 6.1, che, anche se non è detto che dia il risultato ottimo, dà in genere un risultato soddisfacente.

```
p[1] ← 1;
var used:array[1...n] of boolean;
for i ← 1 to n do used[i] ←false;
for i ← 2 to n do
begin
  trova j > 1 t.c.  used[j] =false
  e sia minimo dist(m[p[i − 1]], m[j]);
  p[i] ← j;
  used[i] ←true;
end
```

Tabella 6.1: Euristica percorso minimo (pseudocodice).

All'inizio si parte con p_1 (sink) e in ogni iterazione si trova chi è il nodo più vicino, tra quelli non ancora selezionati, e lo si aggiunge al percorso.

Ad esempio supponiamo di avere una rete di sensori composta da 7 nodi dotati di termometro, più il nodo sink, disposti come nella figura 6.1. Eseguendo la query

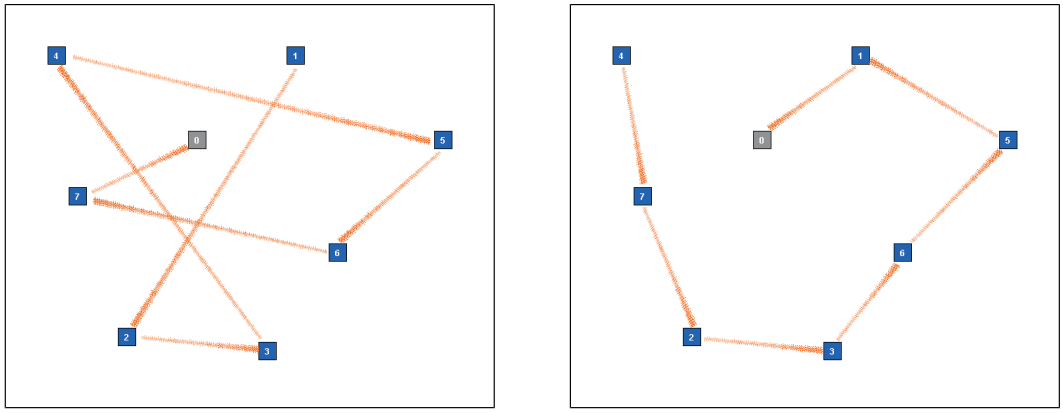


Figura 6.1: Risultato dell'ottimizzazione topologica eseguendo la query "`SELECT Temperature FROM AVG(ALL.Temperature)`" su una rete di sette nodi, più il sink (nodo 0, colorato di grigio). Sono mostrate le comunicazioni effettuate con l'ottimizzazione disabilitata (a sinistra) e abilitata (a destra).

"`SELECT Temperature FROM AVG(ALL.Temperature)`" per ottenere la temperatura media dell'intera rete, l'algoritmo di percorso minimo farà comunicare i nodi come illustrato nella parte destra della figura. La media verrà calcolata nella rete stessa, mentre i dati compiono il percorso mostrato.

L'ordinamento dei sottoespressioni per nodo di appartenenza ha anche un altro effetto positivo di bilanciamento del carico delle risorse. Infatti se considerano l'espressione $\bowtie^1 \ (\bowtie^2 \ (\bowtie^1 \ (2.A\,,\,1.A\,)\,,\,2.L\,)\,,\,1.L\,)$ si vede che si fanno scambiare fra i nodi 1 e 2 molti messaggi. Invece se si usa l'espressione equivalente $\bowtie^2 \ (\bowtie^2 \ (\bowtie^1 \ (1.A\,,\,1.L\,)\,,\,2.A\,)\,,\,2.L\,)$ si vede che i due nodi si scambiano un solo messaggio, distribuendosi meglio il carico della query. Per ottenere questo effetto non è necessario seguire un percorso minimo, ma solo che le sottoespressioni siano ordinate per nodo con un criterio qualunque. Nell'implementazione si è lasciata l'opzione di effettuare solo questo ordinamento evitando il calcolo del percorso minimo (opzione *Resource Balancing*, vedi fig. 7.2).

6.2 Ottimizzazione del piano di esecuzione

6.2.1 Obiettivi

In questo paragrafo ci occupiamo dell'ottimizzazione del piano di esecuzione. La quale si esegue a partire dal piano di esecuzione iniziale creato dalla query prima di disseminarlo fisicamente nei nodi.

L'obiettivo dell'ottimizzazione è ottenere un piano di esecuzione semanticamente equivalente a quello di partenza ma che offra un costo minore in termini di consumo di energia e/o di utilizzo di risorse.

Due piani di esecuzioni sono semanticamente equivalenti quando i risultati che producono sono gli stessi, cioè si hanno le stesse ennuple in uscita con gli stessi valori e attributi.

Durante questa ottimizzazione si cerca di minimizzare i costi di:

- attivazione, cioè energia spesa per fare acquisizione di dati;
- trasmissione, cioè energia spesa per mandare e ricevere dati da un mote all'altro;
- elaborazione e consumo di risorse.

In proporzione il costo più alto si ha per la trasmissione dei dati, poi si ha quello per l'acquisizione ed infine si ha quello per l'elaborazione. Infatti secondo [31] trasmettere un singolo bit di dati ha un costo uguale a eseguire 800 istruzioni. è prevedibile che, col progredire della tecnologia, i processori migliorano molto più velocemente dell'antenna, rendendo sempre più conveniente la computazione che non la trasmissione.

L'ottimizzazione è ottenuta applicando un certo numero di volte delle regole che trasformano porzioni dell'albero di esecuzione. Ogni regola si può applicare solo se le sottoespressioni in questione soddisfano delle determinate condizioni. Questo garantisce che dopo ogni trasformazione locale il costo globale dell'albero rimane lo stesso o diminuisce (ma non aumenta) e nel frattempo l'espressione rimane equivalente.

Applicando le regole si diminuisce il costo attraverso uno o più dei seguenti modi:

- riducendo il numero di messaggi da inviare;
- riducendo la dimensione di messaggi da inviare;
- riducendo il numero di volte che si attivano i sensori;
- riducendo le risorse usate di memoria e di computazione nei nodi.

Nella sezione 6.2.3 verranno presentate le regole di ottimizzazione e nella sezione 6.2.4 verrà presentato l'algoritmo di ottimizzazione che usa queste regole nell'ordine giusto.

6.2.2 Notazione usata

La notazione usata per scrivere le regole è la seguente:

$$\frac{sottoespressione\ originale}{sottoespressione\ trasformata} \tag{6.2}$$

Le espressioni sono espresse con l'algebra degli stream (vedi par. 3.5). Di seguito riprendiamo gli operatori definiti:

- π_X è la proiezione su un insieme di attributi X;
- σ_c è la selezione sulla condizione c;
- $\bowtie$ è l'operazione di join su un attributo che sarà sempre quello di $Timestamp$. Se la join è di tipo sincronizzato scriveremo $\bowtie_{sync}$ e in tal caso il figlio **destro** sarà uno stream sensore *on demand*;
- avg_a è l'operatore media sull'attributo a (operatore binario) e se esso come prefisso ha $[p]$, allora si tratta di una media parziale,se ha $[f]$ si tratta di media finale e se ha $[p/f]$ il tipo è ininfluente;

- $\max_a$ / $\min_a$ è l'operatore binario *max* o *min* sull'attributo a.

Ogni operatore viene eseguito su un nodo specifico. Questo è denotato associando un'apice all'operatore che indica l'identificatore del nodo su cui esso viene eseguito. Se l'apice non è specificato oppure è un underscore (come ad es. $\bowtie^{\text{-}}$) significa che la localizzazione è ininfluente.

Per identificare sottoespressioni generiche si useranno le tre lettere maiuscole A, B, C.

Mettendo un'apice ad un'espressione, come ad esempio A^h si denota che l'operatore più esterno dell'espressione è eseguito sul nodo h. Una sottoespressione può essere convenientemente rappresentata con un albero e in tal caso l'operatore più esterno è la radice dell'albero.

Se c è una condizione, allora con $Attr(c)$ si indica l'insieme di attributi usati nella definizione di c.

Con $Root(A)$ si intende la radice di A, cioè l'operatore più esterno dell'espressione A.

Con $O(A)$ si intende l'insieme degli attributi contenuti nello stream prodotto in output dall'espressione A.

Quando l'espressione è composta da uno stream sensore (albero composto da una sola foglia), per chiarezza useremo la notazione $\breve{S}$.

Con ξ^* si intende una qualunque sequenza, anche vuota, di operatori unari π o σ eseguiti su qualunque mote e con qualunque condizione (per le restrizioni) o insieme di campi (per le proiezioni).

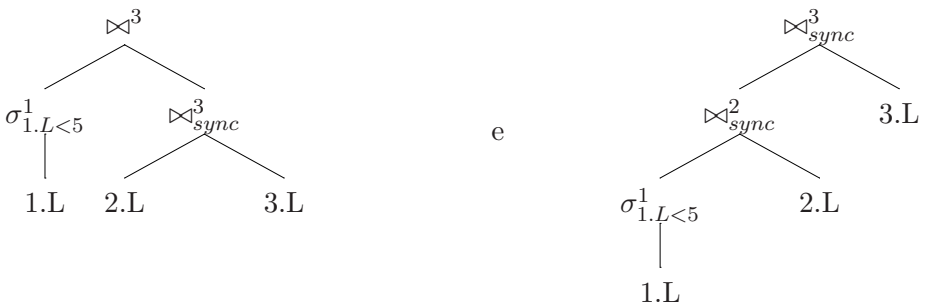
6.2.3 Regole di Ottimizzazione

Sono state introdotte e implementate le seguenti regole di ottimizzazione del piano di esecuzione:

- Ridisposizione dei join:

  quando ci sono tanti join a catena, l'ordine di esecuzione è molto importante per i costi di attivazione. Ad esempio:

  Consideriamo i due piani di esecuzione equivalenti:



  Nel primo albero si campiona sempre la luce in tutti i tre i nodi consumando batteria inutilmente. In realtà è necessario campionare 2 e 3 solo quando

$1.Luce < 5$. Questo è ottenuto tramite l'uso del *sync_join* come indicato nell'albero di destra. Infatti, i join avendo come figli di destra stream sensori, sono di tipo sync e di conseguenza si misura la luce nel nodo 2 e 3 solo quando viene rivelata nel nodo 1 una luce minore di una costante.

I *sync_join* sono più efficienti dei join e quindi bisogna favorire il loro utilizzo. Per il funzionamento è bene avere degli alberi sbilanciati a sinistra, affinché gli stream sensore si avvicinino quanto più possibile ai join e si possano applicare i *sync_join*.

Costruiamo una regola che ci permette di far ciò:

$$\frac{\bowtie^{\text{-}} (A^h, \xi^*(\bowtie^{\text{-}} (B^k, C^j)))}{\widetilde{\xi^{j*}}(\bowtie^j (\bowtie^k (A^h, B^k), C^j))} \tag{6.3}$$

dove $\widetilde{\xi^{j*}}$ è la stessa sequenza di ξ^* con ogni π_X trasformata in $\pi_{X \cup O(A)}$, ed inoltre tutti gli elementi vengono localizzati nel mote j.

In questo modo si ha la possibilità di avere tutti gli stream di destra *ondemand*, così da ridurre i costi d'attivazione.

- Avvicinamento degli operatori unari al nodo che acquisisce (o rilocazione degli operatori unari):

  l'allocazione degli operatori unari, cioè restrizioni, aggregati temporali e/o proiezioni, può essere ottimizzata spostandoli nello stesso nodo dove si acquisiscono i dati. Ad esempio:

  Nel piano di esecuzione non ottimizzato si misura la luce nel nodo 1, e dopo si manda un'ennupla con attributi $\{1.Light, Timestamp\}$ al sink (nodo 0) dove si scarta l'attributo $Timestamp$. Dopo l'ottimizzazione l'attributo $Timestamp$ viene scartato nel nodo di acquisizione e vengono mandati al sink messaggi di 2 bytes invece di 4.

  Si introducono tre regole per applicare ottimizzazioni come quella descritta nell'esempio:

$$\frac{\pi^h_X(A^k) \quad , \quad h \neq k}{\pi^k_X(A^k)} \tag{6.4}$$

$$\frac{\sigma^h_c(A^k) \qquad \text{dove } h \neq k}{\sigma^k_c(A^k)} \tag{6.5}$$

$$\frac{{}_{(Timestamp/(epoch \times p))}\gamma^h_{\{aggr_1(a_1),..,aggr_n(a_n)\}}(A^k) \quad \text{dove } h \neq k}{{}_{(Timestamp/(epoch \times p))}\gamma^k_{\{aggr_1(a_1),..,aggr_n(a_n)\}}(A^k)} \tag{6.6}$$

  In questo modo lo stream tra A e l'operatore unario diventa locale e quindi a costo zero di trasmissione. Inoltre lo stream di output dell'operatore unario

potrebbe diventare remoto avendo comunque dei vantaggi poiché nel caso della restrizione o dell'aggregato temporale si mandano comunque meno messaggi (regola 6.5), e nel caso della proiezione si inviano messaggi più corti (regola 6.4). Quindi in tutti i casi il costo di trasmissione totale diminuisce.

Si noti che come in tutte le altre regole, ogni volta che un operatore unario viene spostato nell'albero o creato, viene già localizzato nello stesso nodo del suo figlio.

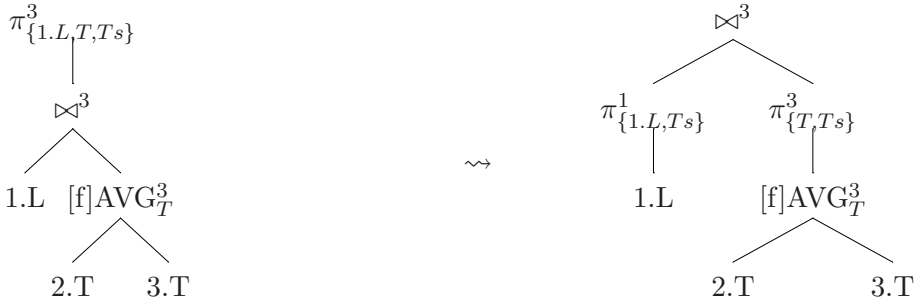
- Anticipare le proiezioni rispetto al join:

quando il join non può diventare di tipo sync, perché ha come figli aggregati, se abbiamo sopra il join una proiezione questa si può semplicemente anticipare per ridurre la dimensione delle ennuple.

Ad esempio data la query:

```
SELECT 1.Light, Temperature, Timestamp
FROM 1.Light, avg(2.Temperature,3.Temperature)
```

si ottiene un'albero logico che si ottimizza così:



L'albero di destra è più conveniente rispetto a quello di sinistra perché la dimensione delle ennuple trasmesse dal mote 1 al mote 3 si riduce grazie alla proiezione sull'insieme di attributi $\{1.L, Ts\}$ e poi la proiezione relativa agli attributi dell'insieme $\{T, Ts\}$, invece consentirà di applicare altre regole di ottimizzazione (ad es. la regola 6.25).

Quando il $Timestamp$ è compreso negli attributi proiettati, si usa dunque la seguente regola:

$$\frac{\pi_X(\bowtie(A^h, B^k)) \qquad \text{dove } Timestamp \in X \text{ e } Root(B) \text{ è un op di aggr}}{\bowtie^k (\pi^h_{X \cap O(A)}(A^h), \pi^k_{X \cap O(B)}(B^k))} \tag{6.7}$$

In questo modo si riduce la quantità di dati da trasmettere al join e di conseguenza il costo di trasmissione nel caso in cui il join sia in una locazione diversa rispetto ad A o B.

Si noti che se $X \cap O(A) = O(A)$ allora $\pi_{X \cap O(A)}$ è ridondante e può essere eliminata (analogamente per $O(B)$).

Quando il $Timestamp$ non è compreso negli attributi proiettati, bisogna ricordarsi che tale attributo è indispensabile per eseguire il join e vale la seguente regola:

$$\frac{\pi_X(\bowtie (A^h, B^k)) \qquad \text{dove } Timestamp \notin X \text{ e } Root(B) \text{ è un op di aggr}}{\pi_X(\bowtie^k (\pi^h_{X \cap O(A) \cup \{Timestamp\}}(A^h), \pi^k_{X \cap O(B) \cup \{Timestamp\}}(B^k)))} \tag{6.8}$$

Infatti i join sono eseguiti sulla uguaglianza dell'attributo $Timestamp$ e quindi deve essere conservato quando si fanno le proiezioni interne. L'attributo $Timestamp$ verrà rimosso solo dal proiezioni esterno che viene appositamente conservata (nella sua locazione originale).

Si noti che in entrambi i casi i nodi π vengono creati nella stessa locazione delle sottoespressioni a cui si riferiscono.

- Anticipare le proiezioni rispetto al sync_join:

l'eliminazione degli attributi non richiesti deve essere fatta prima possibile nel piano di esecuzione perché riduce la dimensione dei messaggi. Tuttavia bisogna far attenzione a non far scendere una proiezione fra una *sync_join* e il suo stream sensore on demand, altrimenti si otterrebbe solo di aumentare i costi di attivazione. Ad esempio, consideriamo la seguente trasformazione di alberi equivalenti:

Questa non è un'ottimizzazione perché il piano di esecuzione trasformato ha costi di attivazione più alti (nel nodo 2 la luce viene campionata tutte le volte invece che soltanto quando la luce nel nodo 1 è minore di 20). Oltre ad essere dannoso l'operatore $\pi^2_{\{2.L,Ts\}}$ è anche inutile perché riduce la dimensione delle ennuple trasmesse in uno stream locale interno al nodo 2.

L'ottimizzazione giusta sarebbe stata quella in cui la proiezione si replica solo a sinistra:

In questi casi non si applicano le regole 6.7 o 6.8. Poiché in esse è stata aggiunta la condizione "*Root*(B) sia un aggregato" proprio per impedirlo. Invece si usano le seguenti regole:

$$\frac{\pi_X(\bowtie(A^h, \breve{S}^k)) \qquad \text{dove } Timestamp \in X}{\pi^k_X(\bowtie^k_{sync}(\pi^h_{X\cap O(A)}(A^h), \breve{S}^k))} \tag{6.9}$$

e

$$\frac{\pi_X(\bowtie(A^h, \breve{S}^k)) \qquad \text{dove } Timestamp \notin X}{\pi^k_X(\bowtie^k_{sync}(\pi^h_{X\cap O(A)\cup\{Timestamp\}}(A^h), \breve{S}^k))} \tag{6.10}$$

Si noti che in entrambi i casi h e k possono essere uguali o diversi.

In questo modo il join si mantiene sync (o lo diventa se non lo era già). Sia la join che la proiezione originale vengono spostati nel nodo k (già presenti oppure no). Il nodo $\breve{S}^k$ non necessita di proiezione (che impedirebbe il sync) perché è nello stesso nodo del padre e quindi lo stream di output è locale. La proiezione esterna serve per rimuovere gli attributi di troppo provenienti da $\breve{S}^k$ (compreso il $Timestamp$).

Si nota che se ci sono delle proiezioni e/o restrizioni nel piano di esecuzione originale tra il π e la $\bowtie$, prima di applicare questa regola bisognerà far scendere la proiezione usando le regole 6.20, 6.21 e 6.22. S deve anche notare che queste regole si applicano indipendentemente dall'allocazione originale del join, la quale infatti non è specificata nella parte superiore della barra della regola (se questa locazione non fosse originalmente k, e quindi il join non fosse di tipo sync lo diventerà comunque dopo l'applicazione della regola).

- Eliminazione delle proiezioni:

un operatore di proiezione che non elimina nessun attributo è inutile ed anche se la relativa rimozione non riduce nessun costo di attivazione o di trasmissione,

semplifica il piano di esecuzione e diminuisce il numero di operatori eseguiti su un nodo e così abbiamo la seguente regola:

$$\frac{\pi_X(A) \qquad \text{dove } X = O(A)}{A} \tag{6.11}$$

- Anticipare le restrizioni rispetto al sync_join:

eseguire le restrizioni prima possibile è in generale conveniente, perché riduce il numero di messaggi inviati. Tuttavia bisogna fare attenzione a non frapporre una restrizione fra un nodo *sync_join* e lo stream sensore on demand, altrimenti il *sync_join* non può più essere usato e si aumentano i costi di attivazione (e la riduzione del numero di ennuple diventa inutile, poiché lo stream il cui traffico si riduce è locale). Le successive due regole consentono di anticipare le restrizioni rispetto ad una *sync_join* e si possono applicare solo quando non disturbano lo stream on demand:

$$\frac{\sigma_c(\xi^*(\bowtie_{Sync}(B^k, \breve{S}^h))) \qquad \text{dove } Attr(c) \subseteq O(B)}{\xi^*(\bowtie^h_{Sync}(\sigma^k_c(B^k), \breve{S}^h))} \tag{6.12}$$

dove ξ^* è una qualunque sequenza, anche vuota di π e σ (eseguiti su qualunque nodo e con qualunque parametro).

Si noti che σ si può sempre anticipare in basso rispetto a tutte le operazioni in ξ^*. In questo modo dopo la trasformazione il nodo join riceve un numero minore di ennuple dal suo figlio sinistro. Se $h \neq k$ questo riduce il numero di dati trasmessi (per questo si alloca σ proprio nel nodo k, qualunque sia la sua allocazione iniziale). Inoltre il join diventa sync se non lo era già e rimane tale se già lo era. Poiché $\breve{S}^h$ dopo il passo di ottimizzazione è comunque stream on demand, il fatto che da sinistra arrivino meno ennuple al join, riduce le richieste di attivazione di $\breve{S}^h$, diminuendo anche i costi di attivazione.

Analogamente vale la seguente regola:

$$\frac{\sigma_c(\xi^*(\bowtie(\breve{S}^h, B^k))) \qquad \text{dove } Attr(c) \subseteq O(B)\ ,\ B \text{ non stream sensore}}{\xi^*(\bowtie^h_{Sync}(\sigma^k_c(B^k), \breve{S}^h))} \tag{6.13}$$

cioè se il nodo B a cui si lega il σ compare inizialmente a destra del join che a sinistra ha uno stream sensore, allora l'ordine dei due figli si inverte per ottenere o preservare la proprietà del join di essere sync. Si può fare l'inversione perché vale la commutativà per le join:

$$\bowtie(A, B) = \bowtie(B, A)$$

- Anticipare le restrizioni rispetto al join:

l'anticipazione delle restrizioni rispetto al join si può fare direttamente quando non abbiamo *sync_join*, facendo scendere la restrizione sul ramo sinistro:

$$\frac{\sigma_c(\xi^*(\bowtie(A^h, B^k))) \qquad \text{dove } Attr(c) \subseteq O(A)\ ,\ A \text{ e } B \text{ non stream sensori}}{\xi^*(\bowtie_k(\sigma_c^h(A^h), B^k))} \tag{6.14}$$

oppure sul ramo destro:

$$\frac{\sigma_c(\xi^*(\bowtie(A^h, B^k))) \qquad \text{dove } Attr(c) \subseteq O(B)\ ,\ root(B) \text{ aggregato spaziale}}{\xi^*(\bowtie_k(A^h, \sigma_c^k(B^k)))} \tag{6.15}$$

Infatti in questo caso il $\bowtie$ non può comunque essere sync, né diventarlo in seguito e quindi non c'è bisogno di accorgimenti. La locazione del σ, qualunque fosse inizialmente e indipendentemente dalla locazione del join, viene spostata sempre nel nodo di appartenenza della sottoespressione appropriata per minimizzare gli eventuali costi di trasmissione.

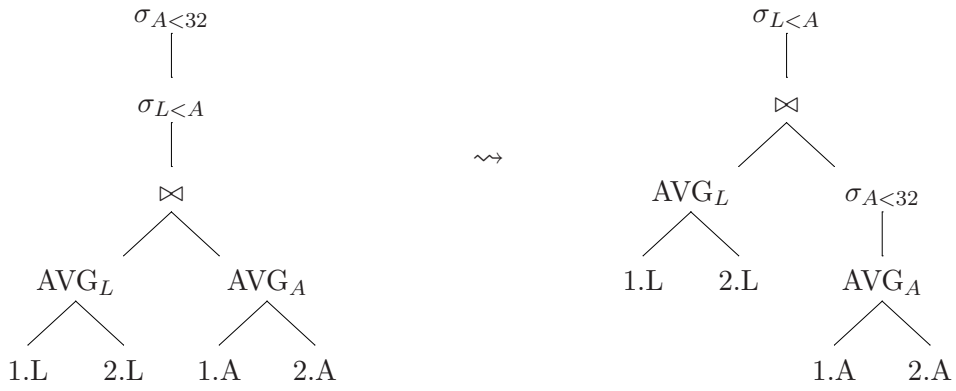
Si osservi che se $Attr(c) \nsubseteq O(B)$ e $Attr(c) \nsubseteq O(A)$ allora questa ottimizzazione non si può applicare. Questo può succedere solo nel caso che la condizione utilizzi due attributi (ad es. $1.Light < 2.Light$) e $O(A)$ e $O(B)$ comprendano ciascuno uno dei due attributi.

Si osservi anche che ξ^* non si può omettere dalle regole. Mostriamo ciò con un esempio.

Data la query:

```
SELECT *
FROM avg(1.Light,2.Light),
     avg(1.Audio,2.Audio)
WHERE Audio<32 AND Light<Audio
```

si ottiene tale piano di esecuzione:



Nell'albero di sinistra l'operatore $\sigma_{L<A}$ è bloccato: non si può farlo scendere applicando la regola 6.15 o 6.14 perché richiede attributi di entrambi i rami. L'altro σ, tuttavia, viene fatto scendere per la regola 6.15 potendo scavalcare il σ bloccato grazie alla presenza dello ξ nella regola.

Un'altro modo per risolvere questo problema potrebbe essere quello di introdurre regole per commutare le restrizioni, ma queste regole sarebbero stati difficili da gestire nell'algoritmo di applicazione delle regole di ottimizzazione (vedi par. 6.2.4).

- Trasformazione del join in sync_join:

quando un join ha a destra degli operatori unari (restrizioni o proiezioni) prima di uno stream sensore, tale stream non è on demand e quindi si attiva il sensore anche quando non c'è bisogno. Questa situazione si può verificare usando le inner query: gli operatori unari sono quelli della inner query. Il problema si può risolvere spostando gli operatori unari sopra il join. Anche se così facendo, si posticipano gli operatori che riducono il traffico di dati dallo stream sensore al join, ciò non incide sul costo perché, allocando il join sullo stesso nodo dello stream sensore, i dati si fanno viaggiare su uno stream locale.

La regola quindi è:

$$\frac{\bowtie^k (B, \xi^*(\breve{S}^h)) \qquad \text{dove } \xi^* \neq \emptyset \vee k \neq h}{\widetilde{\xi^h}(\bowtie^h_{sync} (B, \breve{S}^h))} \tag{6.16}$$

dove $\widetilde{\xi^h}$ è la stessa sequenza di ξ^*, ma con ogni π_X trasformato in $\pi_{X \cup O(B)}$, ed inoltre tutti gli elementi vengono localizzati nel nodo h.

Se lo stream sensore si presenta a sinistra del join, sfruttiamo nuovamente la commmutatività del join, in modo da creare il join di tipo sync:

$$\frac{\bowtie ((\xi^*(\breve{S}^h), B)) \qquad \text{dove } B \text{ non stream sensore}}{\widetilde{\xi^h}(\bowtie^h_{sync} (B, \breve{S}^h))} \tag{6.17}$$

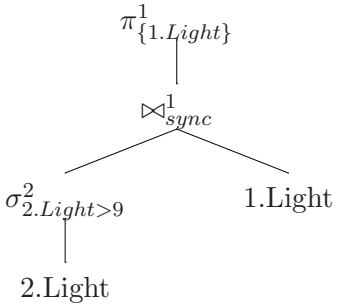
- Eliminazione dei join ridondanti:

qualche volta un'intera sottoespressione è inutile perché gli attributi creati da questa vengono depennati da una proiezione successiva. In questi casi eliminare l'intera sottoespressione riduce i costi di attivazione e trasmissione associati e così si hanno le seguenti due regole:

$$\frac{\pi_X(\bowtie (A, B)) \qquad \text{dove } X \subseteq O(A) \text{ e } B \text{ non contiene nessun } \sigma}{\pi_X(A)} \tag{6.18}$$

$$\frac{\pi_X(\bowtie (A, B)) \qquad \text{dove } X \subseteq O(B) \text{ e } A \text{ non contiene nessun } \sigma}{\pi_X(B)} \tag{6.19}$$

Infatti per la regola 6.19, se $X \subseteq O(B)$ significa che tutti gli attributi che servono sono forniti da quel ramo, mentre l'altro contribuisce solo col $Timestamp$. Qualche volta anche il $Timestamp$ è necessario a eliminare delle ennuple. Ad esempio si consideri il piano di esecuzione:



cioè si richiede il valore della luce nel sensore uno solo quando la luce del sensore due è maggiore di una costante. In questo caso la sottoespressione di sinistra del $sync_join$ non è inutile anche se i suoi attributi non compaiono nell'albero finale, perché serve ad eliminare le ennuple in cui $2.Light \not> 9$. Per questo motivo è stata posta la condizione di non contenere alcuna restrizione nell'albero da eliminare.

- Commutatività della restrizione e della proiezione:

  le proiezioni riducono la dimensione dei messaggi inviati e quindi per ridurre i costi di trasmissione vanno eseguite prima possibile. Le regole (6.7, 6.8, 6.9, 6.10, 6.18, 6.19) devono avere il nodo proiezione a contatto col nodo su cui si applica la regola. è quindi necessario prima di applicarle scavalcare eventuali nodi di restrizione che si frappongono.

  La proiezione sull'insieme di attributi X può essere anticipata rispetto alla restrizione solo se X contiene gli attributi che servono a valutare la condizione della restrizione:

$$\frac{\pi_X(\sigma_c(A)) \qquad \text{dove } Attr(c) \subseteq X}{\sigma_c(\pi_X(A))} \tag{6.20}$$

  Se X non contiene gli attributi usati nella definizione della condizione, si può duplicare la proiezione sotto alla restrizione aggiungendo ad X gli attributi necessari:

$$\frac{\pi_X(\sigma_c(A)) \qquad \text{dove } Y = Attr(c), Y \not\subseteq X \text{ e } O(A) \neq X \cup Y}{\pi_X(\sigma_c(\pi_{X \cup Y}(A)))} \tag{6.21}$$

  La condizione $O(A) \neq X \cup Y$ è una regola anticiclo, per evitare di ripetere questa ottimizzazione più volte sullo stesso nodo (vedi par. 6.2.4).

- Raggruppamento di proiezioni:

  se due proiezioni sono una dietro l'altra, possono essere sostituite da una sola proiezione. Questo di per sé non riduce né il costo di attivazione né quello di trasmissione, ma semplifica il numero di operatori eseguiti in un nodo, che potrebbe essere limitato. Inoltre può consentire successivamente altre ottimizzazioni. Vale la seguente regola:

$$\frac{\pi_X(\pi_Y(A))}{\pi_{X\cap Y}(A)} \tag{6.22}$$

Si noti che affinché l'albero originale sia corretto deve essere che $X \subseteq Y$ e quindi $X \cap Y = X$. Questa regola non cambia le allocazioni.

- Anticipazione della proiezione rispetto a MAX/MIN spaziali:

  la π_X può essere anticipata rispetto ai *max* e ai *min* spaziali. Visto che questi operatori per essere eseguiti hanno bisogno dell'attributo $Timestamp$, si distinguono i casi in cui tale attributo sia presente o no tra quelli proiettati:

$$\frac{\pi_X(\max_a / \min_a(A^h, B^k)) \qquad \text{dove } Timestamp \in X}{\max_a / \min_a(\pi^h_{X'}(A^h), \pi^k_{X''}(B^k))} \tag{6.23}$$

$$\frac{\pi_X(\max_a / \min_a(A^h, B^k)) \qquad \text{dove } Timestamp \notin X}{\pi_X(\max_a / \min_a(\pi^h_{X'\cup\{Timestamp\}}(A^h), \pi^k_{X''\cup\{Timestamp\}}(B^k)))} \tag{6.24}$$

  dove $X' = X$ con prefissi aggiunti ad ogni attributo prendendoli dall'attributo corrispondente in $O(A)$ e X'' analogamente con $O(B)$. Sia X' che X'' includono l'attributo a. L'applicazione della regola serve a diminuire la dimensione dei dati verso il nodo max/min, utile quando questi dati viaggiano su uno stream remoto.

  Per evitare di ripetere questa ottimizzazione più volte sullo stesso nodo imporremo ulteriori condizioni alla regola 6.24 (vedi par. 6.2.4).

- Anticipazione della proiezione rispetto alle AVG spaziali:

  la π_X può essere anticipata rispetto alla *avg* sia parziali che finali, distinguendo i casi in cui gli attributi proiettati includono il $Timestamp$:

$$\frac{\pi_X([p/f]avg_a(A^h, B^k)) \qquad \text{dove } Timestamp \in X}{[p/f]avg_a(\pi^h_{X'}(A^h), \pi^k_{X''}(B^k))} \tag{6.25}$$

$$\frac{\pi_X([p/f]avg_a(A^h, B^k)) \qquad \text{dove } Timestamp \notin X}{\pi_X([p/f]avg_a(\pi^h_{X'\cup\{Timestamp\}}(A^h), \pi^k_{X''\cup\{Timestamp\}}(B^k)))} \tag{6.26}$$

  dove $X' = X$ con prefissi aggiunti ad ogni attributo prendendoli dall'attributo corrispondente in $O(A)$ e X'' analogamente con $O(B)$. Sia X' che X'' includono l'attributo a. In più ad X' si aggiunge l'attributo $\sharp$ (molteplicità) se tale attributo compare nello stream di uscita di $O(A)$ (perché la *avg* in questo caso ne ha bisogno), mentre se $\sharp \notin O(A)$ ma $\sharp \in X$, allora $\sharp$ si rimuove da X' poiché in questo caso $\sharp$ è ottenuto con le ennuple prodotte nel nodo *avg*. Analogamente per X'' e $O(B)$.

  Anche in questo caso lo scopo dell'ottimizzazione è ridurre il traffico dei dati.

6.2.4 Algoritmo di applicazione delle regole di ottimizzazione

La query in MW-SQL viene tradotta in un piano di esecuzione con più join di tipo sync possibile. Infatti ogni join viene allocato nel nodo di allocazione del suo figlio destro e si cerca di rendere i figli destri dei join stream sensori. Le regole di ottimizzazione sono progettate in modo da conservare sync i nodi join coinvolti e per raggiungere questo scopo alcune regole invertono, quando necessario, l'ordine dei figli delle join. Ogni regola di ottimizzazione può solo diminuire i costi di trasmissione e di attivazione del piano di esecuzione.

Un semplice algoritmo di applicazione delle regole di ottimizzazione consiste nell'applicare tutte le regole possibili finché ci sono regole applicabili.

```
Ottimizzazione query:
repeat
   ottimizzazionefatta←false
   ∀ operatore j dell'espressione corrente
     A sottoespressione di radice j
     ∀ regola di ottimizzazione k
       se (k si può applicare ad A)
       allora begin
         applica regola k ad A
         ottimizzazionefatta←true
         esci dai cicli su j e k
       end
until not ottimizzazionefatta
```

Tabella 6.2: Algoritmo di ottimizzazione delle query

Lo pseudocodice è mostrato nella tabella 6.2. Il ciclo sul nodo j si fa visitando l'albero top_down (depth first). Il ciclo sulle regole k significa che tutte le regole potenzialmente applicabili a operatori del tipo di j vengono testate una dopo l'altra. La verifica di applicabilità di una regola si fa verificando se la sottoespressione soddisfa tutte le condizioni della regola. L'applicazione di ogni regola cambia il piano di esecuzione localmente (togliendo, creando e spostando nodi nel piano di esecuzione, e cambiando la loro allocazione nei relativi nodi).

Ad esempio consideriamo la regola (6.7) relativa al caso "Anticipare le restrizioni rispetto al join", cioè:

$$\frac{\pi_X(\bowtie(A^h, B^k)) \qquad \text{dove } Timestamp \in X \text{ e } Root(B) \text{ sia un aggregato}}{\bowtie^k (\pi^h_{X \cap O(A)}(A^h), \pi^k_{X \cap O(B)}(B^k))} \quad (6.27)$$

Quando il nodo j visitato durante la ricerca è di tipo proiezione, per poter applicare la regola si controlla che:

1. il figlio k di j sia di tipo join;
2. il figlio destro di k sia un aggregato spaziale;
3. che l'attributo $Timestamp$ compaia nella lista di attributi di j.

Se almeno una di queste condizioni non è verificata, si passa a controllare le condizioni di un'altra regola per j, controllando quelle che, come la 6.7 hanno come radice dell'albero sopra la barra una proiezione (quindi le regole 6.4, 6.8, 6.9, 6.10, 6.11, 6.18, 6.19, 6.20, 6.21, 6.22, 6.23, 6.24, 6.25, 6.26). Se nessuna risulta applicabile si passa al prossimo nodo della visita e così via.

Se invece, tutte le condizioni della regola sono verificate per j allora essa si applica, cioè si costruisce il piano di esecuzione $\bowtie^k (\pi^h_{X \cap O(A)}(A^h), \pi^k_{X \cap O(B)}(B^k))$ che compare sotto la barra nella regola e lo si sostituisce al piano di esecuzione di radice j. La visita si interrompe e segniamo in un apposito booleano, `ottimizzazionefatta`, che una regola applicabile è stata trovata e applicata. Questo booleano si usa nella guardia del `repeat-until` dello pseudo codice.

Per verificare le condizioni di alcune regole è necessario sapere l'insieme di attributi in output di ogni nodo. Questo insieme è calcolato ricorsivamente per ogni nodo durante il typechecking dell'albero (vedi par. 5.4), e va ricalcolato dopo ogni ottimizzazione per il ciclo successivo.

Esempi dell'esecuzione di questo algoritmo saranno mostrati nel paragrafo 7.4.

Condizioni anticiclo

L'algoritmo descritto sopra ha il problema che potrebbe non terminare.

Infatti le regole 6.8, 6.9, 6.10, 6.21, 6.24 e 6.26 sono ripetibili, cioè possono essere applicate infinite volte a partire dallo stesso nodo j. In tutte queste regole si ha il problema che un operatore proiezione viene replicato e nell'albero risultante si potrebbe riapplicare la stessa regola alla radice e così via, creando una successione di proiezione inutili sotto a j. Questo si può risolvere aggiungendo alle condizioni delle regole precedentemente viste, un'apposita *condizione anticiclo* la quale impedisce di applicare la regola se la proiezione che verrebbe creata è inutile. Una proiezione è inutile se non elimina nessun attributo, cioè il suo insieme di attributi X coincide con l'output del nodo a cui si applica la proiezione.

Si noti che la condizione anticiclo non può mai essere soddisfatta in un nodo k per una data regola dopo che questa è stata già applicata a quel nodo. Qualche volta la condizione anticiclo impedisce l'applicazione di una data regola ad un nodo k, anche se quella regola non è stata ancora mai applicata al nodo k. In questi casi giustamente la regola non viene applicata perché le proiezioni che si creerebbero sarebbero superflue.

Le regole 6.8, 6.24 e 6.26 creano due nuovi operatori π sotto al nodo a cui vengono applicate. La condizione anticiclo per queste regole è che le due proiezioni non siano entrambi inutili. Se una è utile e l'altra inutile la regola si applica lo stesso e poi quella inutile verrà eliminata successivamente con la regola 6.11.

A parte le regole sopra citate, le altre non creano problemi di cicli infiniti perché una volta applicate non possono essere applicate allo stesso operatore dell'espressione risultante. Quindi l'algoritmo Ottimizzazione Query quindi, con le condizioni anticiclo, termina sempre.

Problema dei minimi locali

L'algoritmo Ottimizzazione Query sceglie sempre la prima regola applicabile che trova, perché ogni regola migliora sempre i costi o al massimo li lascia inalterati.

Quindi si tratta di un algoritmo di ricerca locale.

Alcune volte tuttavia, sarebbe importante scegliere bene quale ottimizzazione applicare tra quelle possibili (anche se la maggior parte delle volte non è necessario). Questo può succedere quando abbiamo un join con due stream sensori come figli. Per esempio, a partire dal piano di esecuzione

$$A_0 = \sigma_{1.Light<6}(\sigma_{2.Light>=2}(\xi^*(\bowtie_{sync}(1.Light, 2.Light))))$$

applicando la regola 6.12 al primo σ si può ottenere

$$A_0 \rightsquigarrow A_1 = \sigma_{2.Light>=2}(\xi^*(\bowtie_{sync}(\sigma_{1.Light<6}(1.Light), 2.Light)))$$

oppure applicando la regola 6.13 al secondo σ possiamo ottenere

$$A_0 \rightsquigarrow A_2 = \sigma_{1.Light<6}(\xi^*(\bowtie_{sync}(\sigma_{2.Light>=2}(2.Light), 1.Light)))$$

In entrambi i casi, una volta applicata una regola, l'altra non si può più applicare perché romperebbe la $sync_join$ che diventerebbe m_join. Si noti che i piani di esecuzione A_1, A_2 hanno costi di attivazione e di trasmissione diversi[1], e che non è possibile sapere qual'è meglio senza calcolare i costi (compresi i fattori di selettività delle condizioni).

6.2.5 Classificazione delle regole di ottimizzazione

Regola	Eq.n.	A	B	C
Ridisposizione dei join	(6.3)		✓	
Avvicinam. operat. unari al nodo che acquisisce	(6.4) (6.5) (6.6)	✓		
Antic. le proiezioni rispetto al join	(6.7) (6.8)	✓		
Antic. le proiezioni rispetto al sync_join	(6.9) (6.10)	✓		
Eliminazione delle proiezioni	(6.11)			✓
Anticipare le restrizioni rispetto al sync_join	(6.12) (6.13)	✓	✓	
Anticipare le restrizioni rispetto al join	(6.14) (6.15)	✓		
Trasformaz. join in sync_join	(6.16) (6.17)		✓	
Eliminazione join ridondanti	(6.18) (6.19)	✓	✓	✓
Raggruppamento di proiezioni	(6.22)			✓
Commutatività della restrizione e della proiezione	(6.20) (6.21)	✓		
Anticip. della proiez. rispetto alle MAX/MIN spaz.	(6.23) (6.24)	✓		
Anticip. della proiezione rispetto alle AVG spaz.	(6.25) (6.26)	✓		

Tabella 6.3: Obiettivi raggiunti dall'applicazione delle regole di ottimizzazione.

Le regole che abbiamo descritto nel paragrafo 6.2.3 possono essere classificate a secondo degli obiettivi che raggiungono:

A) minimizzazione dei costi di trasmissione (meno records spediti o record più corti)

[1] Anche molto diversi, come osservato anche dal gruppo di ricerca di Berkeley in [29], nel paragrafo 4.2

B) minimizzazione dei costi di attivazione dei sensori

C) minimizzazione delle risorse usate (il numero di operatori da eseguire su ogni nodo)

Alcune regole ovviamente, raggiungono più obiettivi contemporaneamente. Gli obiettivi raggiunti dalle regole sono riassunti nella tabella 6.3.

Nell'implementazione è stata lasciata la possibilità per l'utente di selezionare solo alcuni dei tre obiettivi anziché tutti. Quando si cercano le regole da applicare (vedi par. 6.2.4) si ignorano le regole che, secondo la tabella 6.3, non contribuiscono ad almeno un obiettivo di quelli prefissati.

Capitolo 7

Realizzazione ed esempi

Mostreremo come sono state realizzate e testate le soluzioni descritte nei capitoli precedenti. Verranno mostrate l'interfaccia e alcuni esempi di esecuzione.

L'implementazione è integrata come modulo del sistema MaD-WiSe (vedi cap. 3).

Lo sviluppo è stato eseguito in Java, quindi il bytecode risultante può essere eseguito su diverse piattaforme.

7.1 Strumenti usati

Lo sviluppo è stato eseguito in **Java** [22], quindi il bytecode risultante può essere eseguito in qualunque piattaforma che abbia una Java Virtual Machine.

Il parser del nuovo linguaggio MW-SQL (vedi capitolo 4) è stato costruito utilizzando "Java Compiler Compiler" (**JavaCC ver 3.2**) [25] un generatore open-source di parser top-down per Java. Data una definizione dei token usati e la grammatica del linguaggio MW-SQL, JavaCC costruisce delle classi per l'analizzatore sintattico e lessicale che riconosce il linguaggio. Quando un simbolo non terminale è riconosciuto viene eseguito del codice per la rappresentazione del piano di esecuzione di una query (vedi fig. 5.2 per un esempio). Infine nel file abbiamo specificato alcune opzioni del parser da generare, come il riconoscimento non case sensitive delle parole riservate e il lookahead di 5 token.

L'interfaccia utente, anch'essa integrata in quella di MaD-WiSe, è stata costruita usando la libreria **swing**.

Per fare testing e debugging del nostro sistema abbiamo usato, per la parte di query processor di MaD-WiSe, sia nodi reali (descritti nel par. 2.1) che l'ambiente di simulazione **TOSSIM**, fornito insieme al sistema operativo TinyOS. TOSSIM [28] è un simulatore ad eventi che consente di effettuare il debugging e lanalisi di un'applicazione per TinyOS, eseguendola su una piattaforma PC, anziché su l'hardware del sensore. Naturalmente TOSSIM non simula il mondo reale, ma fa una serie di semplificazioni e utilizza delle variabili aleatorie per simulare i risultati di campionamenti, senza pregiudicarne tuttavia l'accuratezza dei risultati delle successive computazioni.

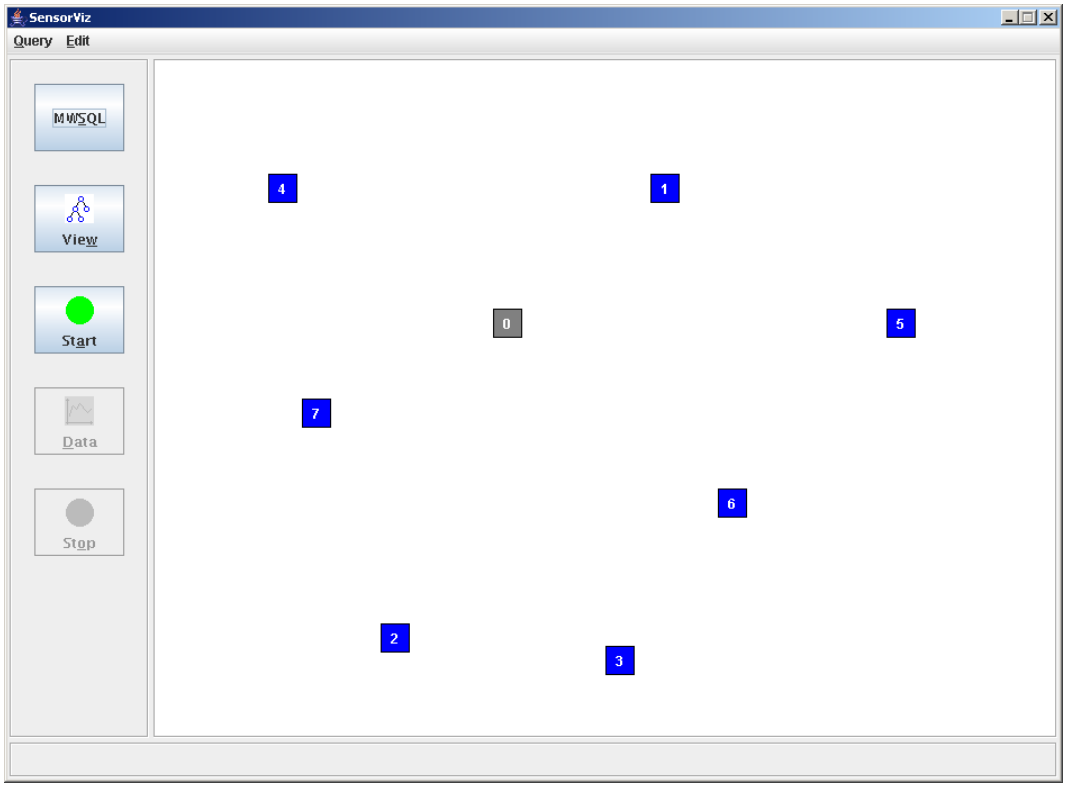


Figura 7.1: Il riquadro principale di MUI, l'interfaccia di MaD-WiSe, nella versione corrente.

7.2 Interfaccia

Il componente che abbiamo aggiunto all'interfaccia consiste in una finestra di dialogo per immettere query che compare premendo il pulsante, etichettato MWSQL, appositamente aggiunto alla barra degli strumenti dell'interfaccia di MaD-WiSe (vedi par. 3.3). La nuova finestra di dialogo è composta da vari pannelli (vedi fig. 7.2):

- una finestra di testo nella quale l'utente può immettere i comandi in MW-SQL (query e definizioni di sorgenti virtuali);
- pulsanti per richiamare alcune query di esempio predefinite alcune con commenti di spiegazione, per illustrare in maniera facile e intuitiva le varie caratteristiche del linguaggio;
- pulsanti per attivare/disattivare l'ottimizzazione topologica (vedi par. 6.1);
- pulsante per attivare/disattivare l'ottimizzazione del piano di esecuzione (vedi par. 6.2). Se attivata l'utente può selezionare con altri tasti gli obiettivi specifici che l'ottimizzatore deve perseguire (vedi par. 6.2.5).

Quando un comando viene processato il primo errore che eventualmente si verifica (errori lessicali, sintattici, di tipo, uso di sorgenti virtuali non definite ecc.), viene riportato all'utente attraverso un messaggio di errore evidenziato nello status bar

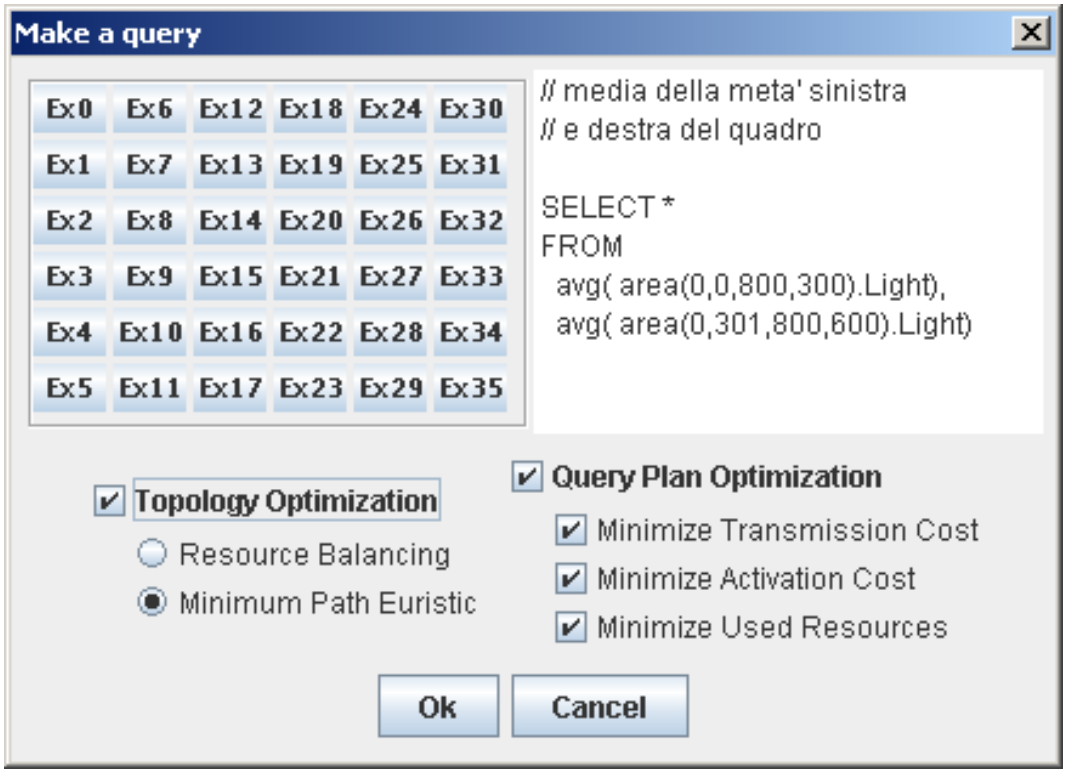


Figura 7.2: La finestra di dialogo per immettere query che compare premendo il pulsante etichettato MWSQL.

di MUI. Quando non si verificano errori la status bar riporta invece il piano di esecuzione risultante.

Se nella finestra di dialogo MW-SQL viene immessa correttamente una definizione di un nuovo sorgente virtuale, allora nello status bar viene riportato l'esito positivo del comando.

Il linguaggio MW-SQL si è rivelato molto più intuitivo ed efficiente da usare per definire query, rispetto al metodo precedentemente usato (creazione manuale di tutti gli stream e operatori necessari nei nodi opportuni). Quest'ultimo è quindi diventato obsoleto e di conseguenza tutti i pulsanti della barra degli strumenti per creare stream e operatori, e i dialoghi corrispondenti sono stati rimossi, sostituendoli con il pulsante MWSQL. L'interfaccia viene così notevolmente semplificata (vedi figg. 3.2 e 7.1).

Il resto dell'interfaccia (cioè i pulsanti Start, Stop, Data, View; il riquadro principale che mostra la rete di sensori nella sua interezza; i riquadri della query che mostrano graficamente per ogni nodo della rete gli stream e gli operatori) è stato mantenuto. In questo modo l'utente dopo aver inserito una query può procedere normalmente, dando l'avvio all'esecuzione e visualizzando i risultati, o anche aprendo i riquadri della query dei vari nodi (col pulsante View) e visualizzando graficamente l'albero di operatori e di stream in cui è stata tradotta la query.

7.3 Struttura del codice

Il codice essendo sviluppato in Java, è strutturato in classi ciascuna dedicata a uno specifico aspetto.

La rappresentazione interna del piano di esecuzione (vedi par. 5.2) è un albero i cui nodi sono istanze di classi diverse, una per ogni tipo di operatore o foglia.

Le classi per rappresentare i nodi sono derivate da una superclasse comune a tutti i nodi che contiene i campi e i metodi condivisi da tutti i tipi (come ad es. la locazione della rete di sensori, l'output rate, l'etichetta di ridenominazione, il riferimento al nodo padre). Le sottoclassi ridefiniscono o aggiungono campi e metodi, che cambiano da operatore a operatore. Si hanno campi per la condizione degli operatori di restrizione, campi per l'insieme di attributi proiettati per l'operatore di proiezione, metodi come quelli definiti diversamente per ogni operatore per trovare la localizzazione iniziale (vedi par. 5.5), metodi per il typechecking, la determinazione degli output e output rate (vedi par. 5.4), l'applicazione delle regole di ottimizzazione del piano di esecuzione (vedi par. 6.2.4) e la traduzione nel piano di esecuzione finale (vedi par. 5.6). Si hanno anche classi aggiuntive per gestire una condizione, un transduttore, un attributo o un'area rettangolare della rete di sensori.

Due ulteriori classi gestiscono gli insiemi di attributi e gli insiemi di sottoalberi. Implementano tutte le operazioni e le relazioni tra insiemi come quelle ad esempio usate nel calcolo delle condizioni delle regole di ottimizzazione o nel calcolo dell'output. La classe di insieme di sottoalberi include anche i metodi per unire i suoi elementi in un'unico albero binario (vedi par. 5.3 e vedi fig. 5.3) eventualmente applicando l'ottimizzazione topologica (vedi par. 6.1).

Una classe è dedicata al mantenimento dello stato corrente del sistema. Questa classe tiene traccia di tutti i sorgenti virtuali definiti in questa sessione o nelle

precedenti (leggendole da un apposito file all'inizio dell'esecuzione del programma) e contiene i dati relativi ai nodi della rete (posizione dei nodi, transduttori presenti, e metadati come la dimensione in byte dei campi). Inoltre tale classe implementa funzioni di utilità associate a questi dati, come ad esempio il calcolo della distanza euclidea fra due nodi della rete utilizzata durante l'ottimizzazione topologica e tiene traccia degli errori generati da tutti gli altri componenti.

Queste classi insieme a quelle create automaticamente dal generatore di parser e a quella della finestra di dialogo di MW-SQL, si integrano con quelle del resto del progetto MaD-WiSe.

7.4 Esempi

In questo paragrafo discutiamo alcuni esempi che mostrano i comportamento dell'ottimizzatore delle query.

QueryA:

```
SELECT 1.Light
FROM 1.Light,2.Light
WHERE 2.Light>10
```

La rappresentazione interna del piano di esecuzione risultante[1] è (dopo l'allocazione iniziale):

$$\pi^1_{\{1.Light\}} \; (\; \sigma^1_{(2.Light>10)} \; (\\ \quad \bowtie^1_{Sync} \; (\; 2.Light \; , \; 1.Light \;) \\ ) \;)$$

questo piano dà il risultato effettivamente voluto (riporta le misurazioni della luce dei nodi 1,2 quando la seconda risulta superiore a 10), tuttavia non e' molto efficiente perché la trasmissione tra i nodi 2 e 1 e il campionamento del nodo 1 si effettuano in ogni caso anche quando la condizione non è soddisfatta.

Attivando tutte le ottimizzazioni, le seguenti regole le cui precondizioni sono soddisfatte vengono trovate e applicate:

Commutatività della restrizione e della proiezione (con duplic.) (regola 6.21):

$$\pi^1_{\{1.Light\}} \; (\; \sigma^1_{(2.Light>10)} \; (\\ \quad \pi^1_{\{2.Light,1.Light\}} \; (\; \bowtie^1_{Sync} \; (\; 2.Light \; , \; 1.Light \;) \;) \\ ) \;)$$

Antic. restrizione rispetto al sync_join (sx) (regola 6.12):

$$\pi^1_{\{1.Light\}} \; (\; \pi^1_{\{2.Light,1.Light\}} \; (\\ \quad \bowtie^1_{Sync} \; (\; \sigma^2_{(2.Light>10)} \; (\; 2.Light \;) \; , \; 1.Light \;) \\ ) \;)$$

[1] La notazione usata è quella descritta nel paragrafo 6.2.2

Raggruppamento di proiezioni (regola 6.22):

$$\pi^1_{\{1.Light\}} \left(\bowtie^1_{Sync} \left(\sigma^2_{(2.Light>10)} \left(2.Light \right), 1.Light \right) \right)$$

Antic. proiezioni rispetto al sync_join (con duplicazioni) (regola 6.10):

$$\pi^1_{\{1.Light\}} \left(\bowtie^1_{Sync} \left(\pi^2_{\{Timestamp\}} \left(\sigma^2_{(2.Light>10)} \left(2.Light \right) \right), 1.Light \right) \right)$$

dopo questi quattro passaggi, l'ottimizzatore non trova altre regole applicabili a partire da nessun nodo, quindi la rappresentazione interna risultante è l'ultima mostrata. A questo punto si procede a costruire il piano finale che può essere visualizzato dall'utente attraverso l'interfaccia (vedi fig. 7.3).

Nell'esempio della queryA si può notare che l'ottimizzatore trova la soluzione ottima (in questo caso): infatti lo stream sensore luce del nodo 1 e' on demand e verrà attivato solo quando la luce del nodo 2 risultera' superiore al valore soglia, la proiezione interna del nodo 2 minimizza la quantità di dati trasmessi da questo nodo al nodo 1 e infine la proiezione esterna minimizza la quantità di dati trasmessi dal nodo 1 al nodo sink. Tuttavia questa soluzione è stata trovata in quattro passaggi invece che nei due minimi indispensabili.

Un altro esempio è il seguente:

QueryB:

```
SELECT 3.Light, 1.Light
FROM (
       SELECT *
       FROM 2.Light,3.Light
       WHERE 2.Light<10
),1.Light
```

La rappresentazione interna del piano di esecuzione risultante è:

$$\pi^2_{\{3.Light,1.Light\}} \left(\bowtie^2 \left(1.Light, \sigma^2_{(2.Light<10)} \left(\bowtie^2_{Sync} \left(3.Light, 2.Light \right) \right) \right) \right)$$

In questo piano tutti e tre i trasduttori utilizzati sono sempre attivati ad ogni passo di campionamento, anche quando nessuna ennupla verrà prodotta in uscita. Infatti i trasduttori dei nodi 1 e 3 sono periodici, e quello del nodo 2, pur essendo on demand (è il figlio destro di una giunzione sincrona), riceverà sempre richieste di attivazione. Inoltre ad ogni passo di campionamento si effettuerà una trasmissione fra i nodi 3 e 2, e fra 1 e 2.

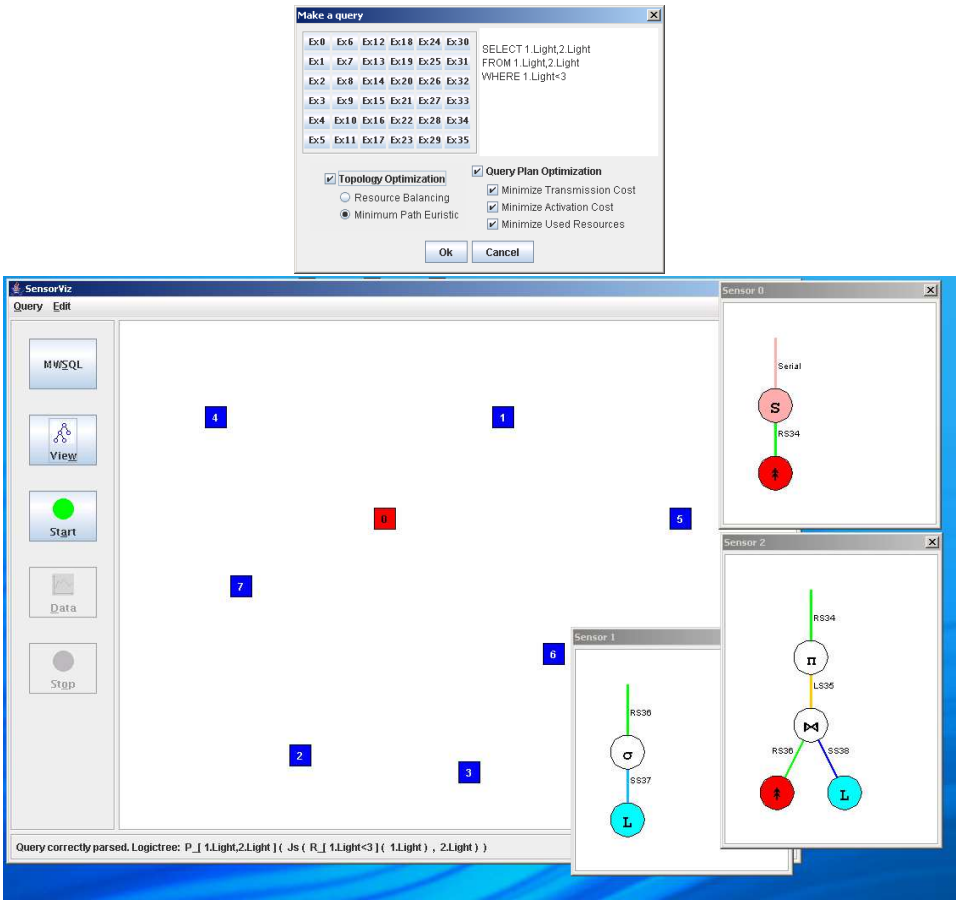


Figura 7.3: Immettendo nella finestra di dialogo la *QueryA* in MW-SQL (in alto), il sistema costruisce il piano di esecuzione ottimizzato composto da operatori e stream in ogni nodo visibili attraverso l'interfaccia (in basso).

L'ottimizzazione, in questo caso, applica al piano iniziale cinque regole in successione:

Trasformaz. join in sync_join (regola 6.16):

$$\pi^2_{\{3.Light,1.Light\}} (\bowtie^1_{Sync} (\sigma^2_{(2.Light<10)} (\bowtie^2_{Sync} (3.Light , 2.Light)) , 1.Light))$$

Antic. le proiezioni rispetto al sync_join (con duplic.) (regola 6.10):

$$\pi^1_{\{3.Light,1.Light\}} (\bowtie^1_{Sync} (\pi^2_{\{Timestamp,3.Light\}} (\sigma^2_{(2.Light<10)} (\bowtie^2_{Sync} (3.Light , 2.Light))) , 1.Light))$$

Anticipare le restrizioni rispetto al sync_join (regola 6.12):

$$\pi^1_{\{3.Light,1.Light\}} (\bowtie^1_{Sync} (\pi^2_{\{Timestamp,3.Light\}} (\bowtie^3_{Sync} (\sigma^2_{(2.Light<10)} (2.Light) , 3.Light)) , 1.Light))$$

Antic. le proiezioni rispetto al sync_join (con dupl.) (regola 6.10):

$$\pi^1_{\{3.Light,1.Light\}} (\bowtie^1_{Sync} (\pi^3_{\{Timestamp,3.Light\}} (\bowtie^3_{Sync} (\pi^2_{\{Timestamp\}} (\sigma^2_{(2.Light<10)} (2.Light)) , 3.Light)) , 1.Light))$$

Eliminazione delle proiezioni (regola 6.11):

$$\pi^1_{\{3.Light,1.Light\}} (\bowtie^1_{Sync} (\bowtie^3_{Sync} (\pi^2_{\{Timestamp\}} (\sigma^2_{(2.Light<10)} (2.Light)) , 3.Light) , 1.Light))$$

la rappresentazione interna risultante è l'ultima mostrata. Il piano di esecuzione finale può essere visualizzato dall'utente attraverso l'interfaccia (vedi fig. 7.4). Si noti che il piano risultante dalle ottimizzazione è molto conveniente dal punto di vista del risparmio energetico. Infatti due stream sensori su tre sono on demand e quindi la loro attivazione è subordinata al superamento del valore soglia della restrizione da parte della misurazione nel nodo 2. Inoltre le due proiezioni rimuovono dati, appena diventano inutili, prima che questi debbano venire trasmessi.

Altro esempio è il seguente:

QueryC:

```
// Media di medie
SELECT *
FROM avg(
    avg(1.Audio,5.Audio,7.Audio),
    avg(2.Audio,3.Audio)
)
```

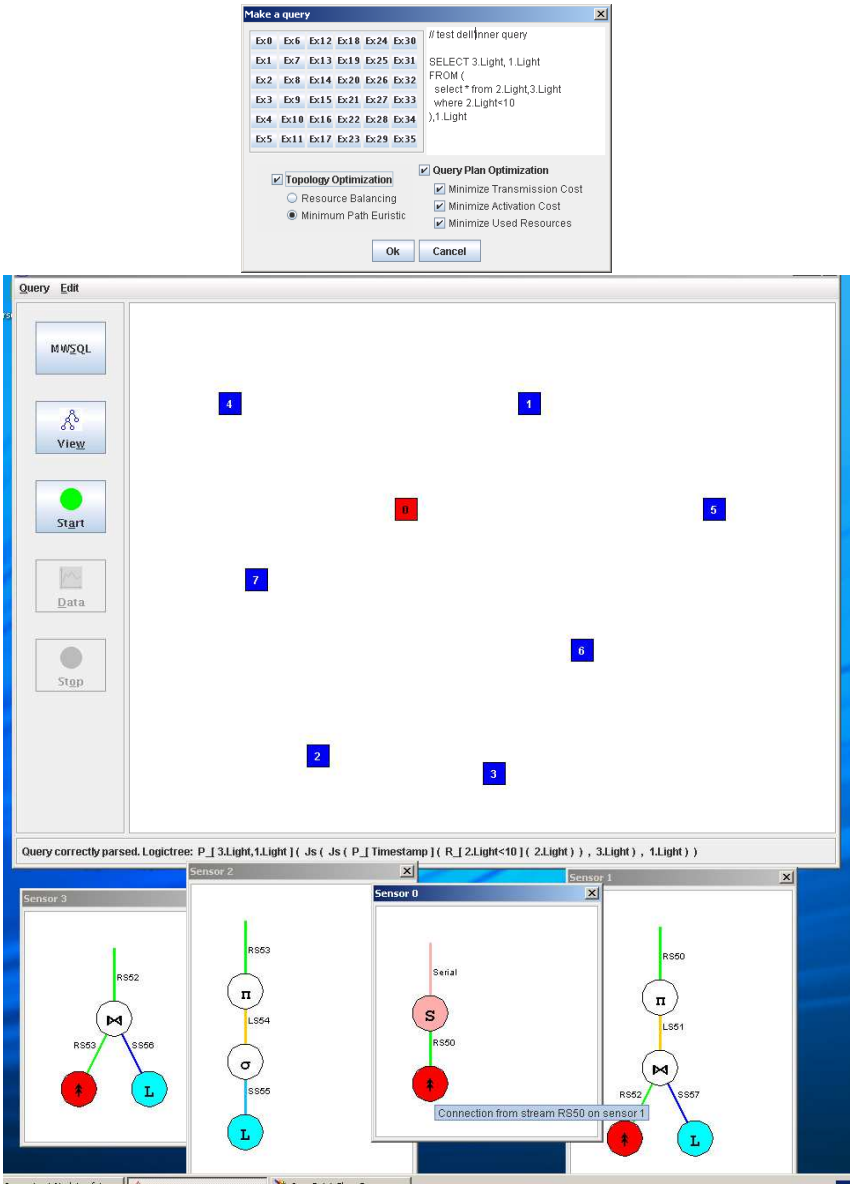


Figura 7.4: Esempio di costruzione del piano di esecuzione ottimizzato per la *QueryB*.

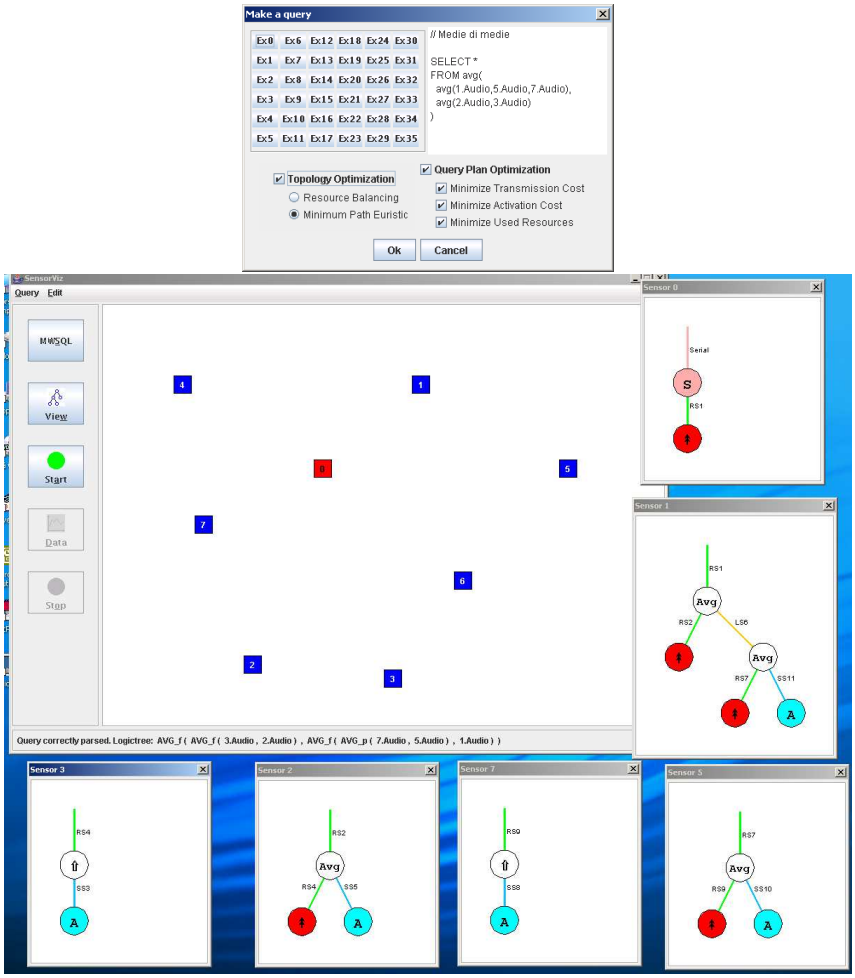


Figura 7.5: Esempio di costruzione del piano di esecuzione ottimizzato per la *QueryC*.

In questo caso non viene effettuata nessuna ottimizzazione e la rappresentazione interna del piano di esecuzione risultante

$$
\begin{aligned}
&[f]avg^{1}_{Audio}\ (\\
&\quad [f]avg^{2}_{Audio}\ (\ 3.Audio\ ,\ 2.Audio\ )\ ,\\
&\quad [f]avg^{1}_{Audio}\ (\ [p]avg^{5}_{Audio}\ (\ 7.Audio\ ,\ 5.Audio\ )\ ,\ 1.Audio\ )\\
&)
\end{aligned}
$$

è direttamente tradotta nel piano di esecuzione finale nella figura 7.4. In questo caso gli operatori di media parziale e finale vengono usati per calcolare l'avg nella rete stessa man mano che i dati la attraversano.

In questo esempio:

QueryD:

```
SELECT 1.Light, 1.Audio
FROM 1
```

l'utente sta chiedendo le misurazioni del fotometro e del microfono del nodo 1. Per come abbiamo definito il linguaggio MW-SQL, nella rappresentazione interna risultante la proiezione è effettuata sulla giunzione degli stream sensori di tutti i transduttori presenti nel nodo 1:

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (1.Light , 1.Temperature) , 1.Audio) ,) 1.AccelX) , 1.AccelY) , 1.MagnetismX) , 1.MagnetismY))$$

L'ottimizzazione del piano di esecuzione rimuove tutti i campionamenti superflui (in sette passaggi) e lascia soltanto i due richiesti dall'utente:

Eliminazione join ridondanti dx (regola 6.18):

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (1.Light , 1.Temperature) , 1.Audio ,) 1.AccelX) , 1.AccelY) , 1.MagnetismX))$$

Eliminazione join ridondanti dx (regola 6.18):

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (1.Light , 1.Temperature) , 1.Audio ,) 1.AccelX) , 1.AccelY))$$

Eliminazione join ridondanti dx (regola 6.18):

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (1.Light , 1.Temperature) , 1.Audio ,) 1.AccelX))$$

Eliminazione join ridondanti dx (regola 6.18):

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (\bowtie^1_{Sync} (1.Light , 1.Temperature) , 1.Audio)))$$

Antic. le proiezioni rispetto al sync_join (con duplic.) (regola 6.10):

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\pi^1_{\{Timestamp,1.Light\}} (\bowtie^1_{Sync} (1.Light , 1.Temperature)) , 1.Audio))$$

Eliminazione join ridondanti dx (regola 6.18):

$$\pi^1_{\{1.Light,1.Audio\}} (\bowtie^1_{Sync} (\pi^1_{\{Timestamp,1.Light\}} (1.Light) , 1.Audio))$$

ELIMINAZIONE DELLE PROIEZIONI (regola 6.11):

$$\pi^{1}_{\{1.Light,1.Audio\}} \; (\bowtie^{1}_{Sync} \; (\\ 1.Light \; , \; 1.Audio \\ ) \;)$$

Si noti che l'ultima proiezione non viene eliminata perché serve ad eliminare l'attributo *Timestamp* che dopo la giunzione è superfluo. Alla fine risulta un piano di esecuzione minimizzato nei costi di attivazione e nelle risorse utilizzate nei nodi.

Altro esempio:

QueryE:

```
SELECT max(6.Light), avg(3.Temperature)
FROM 6.Light, 3.Temperature
WHERE 6.Light<50 AND 6.Light>10
EPOCH 10 SAMPLES
EVERY 15 SECONDS
```

La queryE genera la seguente rappresentazione interna:

$$_{\{Timestamp/150\}}\gamma^{6}_{\{Max(6.Light),Avg(3.Temperature)\}} \; (\\ \sigma^{6}_{(6.Light<50)} \; (\; \sigma^{6}_{(6.Light>10)} \; (\; \bowtie^{6}_{Sync} \; (\; 3.Temperature \; , \; 6.Light \;) \;) \;) \\ )$$

Secondo questo piano i due transduttori utilizzati vengono prima campionati sempre entrambi, giustapponendo i risultati in una ennupla, e solo successivamente si scartano le ennuple che non soddisfano le condizioni imposte dall'utente nel costrutto `WHERE`. L'ottimizzatore in questo caso applica i seguenti tre passaggi:

ANTICIPARE LE RESTRIZIONI RISPETTO AL SYNC_JOIN (DX) (regola 6.13):

$$_{\{Timestamp/150\}}\gamma^{6}_{\{max(6.Light),avg(3.Temperature)\}} \; (\\ \sigma^{6}_{(6.Light>10)} \; (\; \bowtie^{3}_{Sync} \; (\; \sigma^{6}_{(6.Light<50)} \; (\; 6.Light \;) \; , \; 3.Temperature \;) \;) \\ )$$

ANTICIPARE LE RESTRIZIONI RISPETTO AL SYNC_JOIN (SX) (regola 6.12):

$$_{\{Timestamp/150\}}\gamma^{6}_{\{max(6.Light),avg(3.Temperature)\}} \; (\\ \bowtie^{3}_{Sync} \; (\; \sigma^{6}_{(6.Light>10)} \; (\; \sigma^{6}_{(6.Light<50)} \; (\; 6.Light \;) \;) \; , \; 3.Temperature \;) \\ )$$

AVVICINAM. DELL'OP. AGGR. TEMP AL NODO CHE ACQUISISCE (regola 6.6)

$$_{\{Timestamp/150\}}\gamma^{3}_{\{max(6.Light),avg(3.Temperature)\}} \; (\\ \bowtie^{3}_{Sync} \; (\; \sigma^{6}_{(6.Light>10)} \; (\; \sigma^{6}_{(6.Light<50)} \; (\; 6.Light \;) \;) \; , \; 3.Temperature \;) \\ )$$

Nel piano finale corrispondente è necessario che la temperatura del nodo 6 superi entrambi le restrizioni affinché il termometro del nodo 3 debba essere attivato.

Altri esempi possono essere provati scaricando il codice disponibile in [16].

Capitolo 8

Conclusioni

In questo lavoro abbiamo mostrato come una rete di sensori senza fili possa essere gestita in maniera intuitiva tramite un apposito linguaggio di interrogazione ad alto livello che presenti all'utente la rete stessa come se fosse un database relazionale.

I principali contributi quì presentati sono:

- **definizione di MW-SQL**, un nuovo linguaggio di interrogazione simile ad SQL, per le reti di sensori (cap. 4). Questo linguaggio include costrutti per accedere ai vari transduttori di qualunque nodo, differenziare fra aggregati temporali e spaziali, specificare una sottoarea di interesse della rete, definizione di sorgenti virtuali, determinare le frequenze di campionamento e gli intervalli di tempo su cui effettuare gli aggregati temporali. Le query in MW-SQL vengono tradotte in piani di esecuzione (cap. 5);

- **progettazione e realizzazione di un ottimizzatore di query** (cap. 6). Questo include sia un'ottimizzazione topologica che cerca di minimizzare la lunghezza del percorso dei dati nella rete, sia un'ottimizzazione del piano di esecuzione, caratterizzato dall'avere come obiettivo il risparmio energetico. Le regole di ottimizzazione sfruttano le particolarità del sistema utilizzato, come ad esempio la presenza di operatori misti pull-based e push-based;

- **adattamento dell'interfaccia** (cap. 7) **e dell'algebra su stream** del sistema MaD-WiSe (cap. 3) per riflettere le nuove funzionalità. E' stato possibile semplificare l'interfaccia basandola principalmente su una finestra di dialogo per immettere query in MW-SQL. Il linguaggio MW-SQL ha permesso di definire alcune query la cui esecuzione ha richiesto di ridefinire ed adattare alcuni operatori del sistema MaD-WiSe e aggiungerne altri.

L'implementazione del sistema MaD-WiSe, che include il lavoro presentato in questa tesi, è disponibile alla pagina del omonimo progetto open-source [16].

Il lavoro quì presentato è stato usato in un progetto dell'ISTI/CNR di Pisa in collaborazione con il Comando Provinciale dei Vigili del Fuoco di Pisa (vedi fig. 8.1) per il monitoraggio di parametri fisici attraverso una rete di sensori senza fili applicate su tute di pompieri (cinque nodi per tuta). In questo modo si possono immettere query per rilevare temperatura, luce, umidità, accelerazione, livello di anidride carbonica e tramite queste battito cardiaco, livello di fatica, respiro e posizione (sdraiato o in piedi).

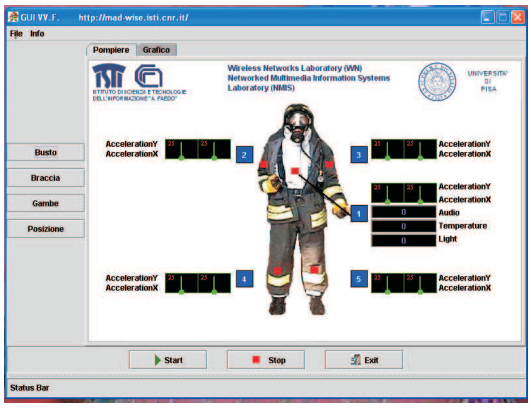


Figura 8.1: Interfaccia grafica di un progetto di monitoraggio di parametri fisici attraverso una WSN applicata su tute di pompieri.

8.1 Sviluppi futuri

Esistono molte linee di sviluppo futuro:

- l'algoritmo usato per scegliere la regola di ottimizzazione da applicare è euristico. Si potrebbe definire un algoritmo basato sul costo dei piani di esecuzione: per calcolare una stima del costo di un piano di esecuzione sarebbe necessario avere a disposizione dati statistici sulle misurazioni e altri metadati;

- si possono aggiungere alcune regole di ottimizzazione per aumentare ulteriormente l'efficienza dell'ottimizzatore. Ad esempio potrebbe essere utile, in alcune condizioni, anticipare un aggregato temporale rispetto all'operatore di giunzione;

- si può estendere l'algebra delle query con l'operatore *unione* e includere nel linguaggio dei nuovi costrutti corrispondenti. Questo sarebbe utile per esempio per formulare query come "dimmi l'identificatore dei nodo dalla temperatura maggiore di un valore soglia" [1];

- estendere l'algebra per rendere possibile il riutilizzo di una misurazione in due computazioni distinte. Questo si potrebbe ottenere con un operatore unario a due stream di output che ricopia le ennuple in ingresso in entrambi i due stream di uscita. Questo renderebbe possibile query come "`SELECT * FROM avg(1.Audio, 2.Audio), avg(1.Audio, 5.Audio)`" che attualmente non sono consentite perché richiedono due misurazioni contemporanee dello stesso transduttore (il microfono del primo nodo);

- aggiungere la possibilità di condividere porzioni del piano di esecuzione fra più query che eseguono contemporaneamente;

[1] Attualmente questo risultato non si ottiene, ad esempio, con query come "`SELECT 1.Light, 2.Light, 3.Light FROM 1,2,3 WHERE 1.Light>50 AND 2.Light>50 AND 3.Light>50`" perché questo restituirebbe solo le ennuple dove *tutte e tre* le temperature sono maggiori di cinquanta.

- verificare la scalabilità del sistema facendo dei test su reti reali con centinaia di nodi o più.

Bibliografia

[1] A. AHO, R. SETHI, and J. ULLMAN, *Compilers: Principles, techniques and tools.*, Addison-Wesley, 1987.

[2] Antonio Albano, *Costruire sistemi per basi di dati*, Addison-Wesley Longman Italia Editoriale SRL, 2001.

[3] Antonio Albano, Giorgio Ghelli, and Renzo Orsini, *Basi di dati relazionali e a oggetti*, Zanichelli Editore S.p.A., 2001.

[4] G. Amato, A. Caruso, S. Chessa, Masi V., and Urpi A., *State of the art and future directions in wireless sensor network's data management*, Tech. Report ISTI-2004-TR-16, ISTI – C.N.R., Pisa, Italy, 2004.

[5] Giuseppe Amato, Paolo Baronti, and Stefano Chessa, *Mad-wise: Programming and accessing data in a wireless sensor networks.*, EUROCON 2005, The International Conference on Computer as a Tool, sponsored by IEEE, Belgrade, Serbia & Montenegro, November 21-24 2005.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom, *Models and issues in data stream systems*, PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (New York, NY, USA), ACM Press, 2002, pp. 1–16.

[7] U.C. Berkeley, *Smart buildings admit their faults*, Lab Notes: Research from the Berkeley College of Engineering - Web Page http://www.coe.berkeley.edu/labnotes/1101smartbuildings.html, 2001.

[8] bernhard Stegmaier, Richard Kuntscheke Lee, and Alfons Kemper, *Streamglobe: Adaptative query processing and optimization in streaming p2p environments*, Proceedings of the 1st workshop pn Data Management for Sensor Network, 2004, pp. 88–97.

[9] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri, *Towards sensor database systems*, MDM '01: Proceedings of the Second International Conference on Mobile Data Management (London, UK), Springer-Verlag, 2001, pp. 3–14.

[10] S. Ceri and G. Pelagatti, *Distributes databases: Principles and systems*, McGraw-Hill, 1984.

[11] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, *Habitat monitoring: Application driver for wireless communications technology*, 2001.

[12] Alberto Cerpa, Jeremy Elson, Deborah Estrin, and Lewis Girod, *Habitat monitoring: Application driver for wireless communications technology*, First ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean, ACM Press, 2001.

[13] Sirish Chandrasekaran and Michael J. Franklin, *Psoup: a system for streaming queries over streaming data.*, VLDB J. **12** (2003), no. 2, 140–156.

[14] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik, *Scalable distributed stream processing.*, CIDR, 2003.

[15] David E. Culler and Wei Hong, *Wireless sensor networks*, Communications of the ACM **47** (2004), no. 6, 30–33.

[16] MaD-WiSe developer team, *Mad-wise web page*, http://mad-wise.isti.cnr.it/, 2004.

[17] TinyDB developer team, *Tinydb web page*, https://telegraph.cs.berkeley.edu/tinydb, 2004.

[18] Lukasz Golab and M. Tamer Özsu, *Issues in data stream management*, SIGMOD Rec. **32** (2003), no. 2, 5–14.

[19] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy, *The platform enabling wireless sensor networks*, Communications of the ACM **47** (2004), no. 6, 41–46.

[20] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister, *System architecture directions for networked sensors*, Architectural Support for Programming Languages and Operating Systems, 2000, pp. 93–104.

[21] Jason L. Hill and David E. Culler, *Mica: A wireless platform for deeply embedded networks*, IEEE Micro **22** (2002), no. 6, 12–24.

[22] Cay S. Horstmann and Gary Cornell, *Java2: i fondamenti*, McGraw-Hill, 2004.

[23] Wen Hu, Archan Misra, and Rajeev Shorey, *Caps: Energy-efficient processing of continuous aggregate queries in sensor networks.*, PerCom, 2006, pp. 190–199.

[24] Crossbow Inc., *Crossbow sensor model*, https:www.xbow.com, 2005.

[25] JavaCC, *The java compiler compiler*, https://javacc.dev.java.net/, 2006.

[26] Donald Kossmann, *The state of the art in distributed query processing*, ACM Computing Survey **32** (2000), no. 4, 422–469.

[27] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo, *Query languages and data models for database sequences and data streams.*, Proc. of Very Large Data Base, (VLDB 04), Springer-Verlag, 2004, pp. 492–503.

[28] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, *Tossim: accurate and scalable simulation of entire tinyos applications*, SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems (New York, NY, USA), ACM Press, 2003, pp. 126–137.

[29] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong, *The design of an acquisitional query processor for sensor networks*, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12 (New York, NY 10036, USA) (ACM, ed.), ACM Press, 2003, pp. 491–502.

[30] Samuel Madden and Michael J. Franklin, *Fjording the stream: An architecture for queries over streaming sensor data.*, ICDE, 2002, pp. 555–566.

[31] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, *Tag: A tiny aggregation service for ad-hoc sensor networks.*, OSDI, 2002.

[32] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman, *Continuously adaptive continuous queries over streams*, SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM Press, 2002, pp. 49–60.

[33] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David E. Culler, *Supporting aggregate queries over ad-hoc wireless sensor networks.*, 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), 20-21 June 2002, Callicoon, NY, USA, IEEE Computer Society, 2002, pp. 49–58.

[34] Julio C. Navas and Michael Wynblatt, *The network is the database: data management for highly distributed systems*, SIGMOD Record (ACM Special Interest Group on Management of Data) **30** (2001), no. 2, 544–551.

[35] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein, *Using state modules for adaptive query processing.*, Proceedings of the 19th International Conference on Data Engineering, March 5-8, Bangalore, India, IEEE Computer Society, 2003, pp. 353–260.

[36] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou, *Distributed query processing on the grid.*, Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings, 2002, pp. 279–290.

[37] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin, *Habitat monitoring with sensor networks*, Communications of the ACM **47** (2004), no. 6, 34–40.

[38] Stratis D. Viglas and Jeffrey F. Naughton, *Rate-based query optimization for streaming information sources*, SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM Press, 2002, pp. 37–48.

[39] Alec Woo, Sam Madden, and Ramesh Govindan, *Networking support for query processing in sensor network*, Communications of the ACM **47** (2004), no. 6, 47–52.

[40] Yong Yao and Johannes Gehrke, *The Cougar approach to in-network query processing in sensor networks*, SIGMOD Record (ACM Special Interest Group on Management of Data) **31** (2002), no. 3, 9–18.

[41] ———, *Query processing in sensor networks*, Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR), 2003.

[42] Vladimir I. Zadorozhny, Panos K. Chrysanthis, and Prashant Krishnamurthy, *A framework for extending the synergy between mac layer and query optimization in sensor networks*, DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks (New York, NY, USA), ACM Press, 2004, pp. 68–77.